University of Zurich[UZH]

Department of Informatics

Scalable Visualization of Large Datasets

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zürich

for the degree of
Doctor of Science (Ph. D.)

by
David Steiner
from Austria

Accepted on the recommendation of

Prof. Dr. Renato Pajarola
Prof. Dr. Oliver Staadt

2018

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 24, 2018

The head of the Ph. D. program in Informatics: Prof. Sven Seuken, Ph.D.

ABSTRACT

An exponential growth of datasets from different fields of science creates the need for scalable visualization systems to display and explore the data interactively. Such datasets include laser scans of architecture or cultural heritage, which can consist of many hundred millions or even billions of points. Other examples include high-resolution X-ray microtomographies of objects that need to be closely examined in a non-destructive manner, in fields like biology, medicine, or anthropology. The resulting volumetric models can capture details in the micrometer range and also often consist of many billions of points.

Visualizing such datasets at interactive frame rates poses a major challenge to the underlying rendering system, as it often means to process gigabytes of data within a time frame of only a few milliseconds. Consequently, there are high demands regarding the system's throughput and latency. These are often met via scaling the system, i.e., allowing it to accomodate more workload.

Strategies for scaling can include making better use of the available resources, e.g., reducing bandwidth requirements and computational costs. A specific example is our volume visualization system that we extended to allow interactive filtering of volume models (e.g., for feature detection or denoising) in the tensor-compressed domain. These filter operations can be performed significantly faster than with comparable approaches, due to reduced computational and bandwidth costs.

More significantly, a visualization system can be scaled by utilizing additional resources within a machine, or additional machines. Especially the latter creates

further challenges, such as additional communication and synchronization overheads as well as load imbalances. For the development of scalable visualization systems, overcoming such load imbalances is critical, especially when facing the unpredictable load often created by user interaction. Similarly, the amount of available resources might fluctuate, if a machine is not dedicated to only a single task, e.g., in the context of virtualization.

We consequently developed a scalable and flexible rendering task partitioning method and associated node affinity model which allow fine-grained implicit dynamic load balancing via a task pulling mechanism. Our method often outperforms traditional load balancing approaches in terms of performance and scalability, especially in the context of unpredictable load and varying compute resources.

Furthermore, we conducted a study in which we in detail examined the scalability of various load balancing methods provided by the Equalizer parallel rendering framework, which our visualization systems are based on. Finally, we also extended the set of utilities provided by the framework, providing diverse features for alleviating tasks like systematically, reproduceably, and automatically evaluating the performance of scalable visualization systems, the collection of data, and using optimized I/O.

KURZFASSUNG

Das exponentielle Wachstum von Datensätzen aus verschiedenen wissenschaftlichen Disziplinen erzeugt einen Bedarf für skalierbare Visualisierungssysteme, um diese Daten interaktiv darstellen und erforschen zu können. Diese sind z.B. Laser-Scans von architektonischen Objekten und Kulturgütern, die oft aus vielen hundert Millionen oder Milliarden von Punkten bestehen. Ein anderes Beispiel sind hochauflösende Mikro-Computertomografien von Objekten, z.B. aus der Biologie, Medizin oder Anthropologie, welche genau untersucht werden sollen, ohne diese zu zerstören. Entsprechende volumentrische Datenmodelle bilden Strukturen im Mikrometerbereich ab und bestehen oft ebenfalls aus vielen Milliarden von Punkten.

Solche Datensätze interaktiv zu visualisieren ist eine grosse Herausforderung für das zugrundeliegende Rendering-System, da dies oft bedeutet Gigabytes an Daten innerhalb von wenigen Millisekunden zu verarbeiten. Entsprechend hoch sind die Anforderungen an Latenz und Datendurchsatz bei solchen Systemen. Diese werden oft durch Skalierung des Systems erreicht, wodurch dieses in die Lage versetzt wird, mehr Arbeitslast zu bewältigen.

Strategien für eine solche Skalierung beinhalten auch eine bessere Nutzung der verfügbaren Resourcen und z.B. die Reduzierung der Menge an benötigter Bandbreite oder Rechenleistung. Ein konkretes Beispiel ist ein Volumen-Visualisierungssystem, das wir erweitert haben um das interaktive Filtern von Volumendaten zu ermöglichen (z.B. zum Hervorheben von Features oder zum Reduzieren von Störsignalen), und zwar in tensor-komprimierter Form. Durch die resultieren-

de Reduzierung von Bandbreiten- und Rechenkosten können solche Filteroperationen deutlich schneller durchgeführt werden, als bei vergleichbaren Ansätzen.

Eine noch deutlichere Skalierung des Systems kann erzielt werden, indem man pro Maschine mehr Ressourcen, oder mehr Maschinen insgesamt nutzt. Dies führt jedoch zu neuen Herausforderungen, wie zusätzliche Kosten für Kommunikation und Synchronisation, sowie auch Ungleichheiten in der Lastverteilung. Das Bewältigen von Letzteren ist kritisch für die Entwicklung von skalierbaren Visualisierungssystemen, insbesondere wenn sich diese mit unverhersehbaren Lasten konfrontiert sehen, die oft durch Benutzerinteraktion hervorgerufen werden. Ähnliches gilt für fluktuierende Ressourcen, was der Fall sein kann wenn eine Machine nicht dediziert an nur einer Aufgabe arbeitet, z.B. im Kontext von Virtualisierung.

Daher haben wir eine Methode für eine skalierbare und flexible Unterteilung von Rendering-Tasks und ein entsprechendes Affinitätsmodell für einzelne Nodes entwickelt, was uns feinkörnige implizite dynamische Lastverteilung über einen Task-Pulling-Mechanismus erlaubt. Unsere Methode ist traditionellen Lastverteilungs-Ansätzen hinsichtlich Leistung und Skalierbarkeit oft überlegen, insbesondere im Kontext von unvorhersehbarer Last und variierenden Ressourcen.

Desweiteren führten wir eine Studie durch, in der wir detailliert die Skalierbarkeit einzelner Lastverteilungsmethoden untersuchen, die vom Equalizer Parallel Rendering Framework angeboten werden, auf welchem unsere Visualisierungssysteme basieren. Schliesslich erweiterten wir auch den Satz an vom Framework angebotenen Werkzeugen um Features, die Aufgaben erleichtern wie die systematische, reproduzierbare und automatische Evaluierung von skalierbaren Visualisierungssystemen, das Sammeln von Daten, und die Verwendung von optimierter I/O.

ACKNOWLEDGMENTS

I would like to thank all the people I had the pleasure to work and collaborate with during the years of my PhD studies, and with whom I had countless interesting and insightful conversations.

My advisor Renato Pajarola I want to especially thank for the great opportunity of pursuing a doctorate, for his advice and patience, and I also thank Oliver Staadt for being my co-advisor.

I want to thank my colleagues and my advisors for their support and for all I had the opportunity to learn from them, especially during our collaborations.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

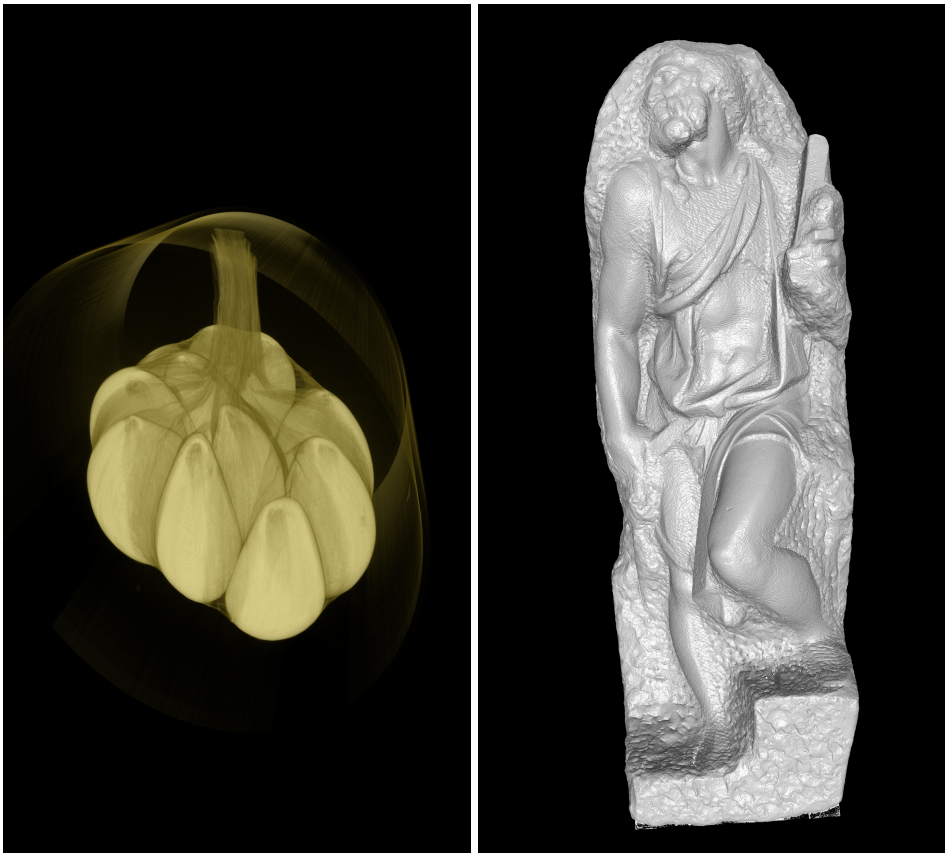
1.1 Visualization

Visualization has always been an invaluable tool for scientists. Attempts at visualizing scientific data to understand and communicate it, e.g., plotting planetary movements over time, can be traced back to the 10th century [Friendly, 2006], predating the scientific revolution and hence being older than the enterprise of modern science itself. However, scientific visualization as a discipline was only established in the 1980s [DeFanti et al., 1989], taking advantage of increasingly powerful supercomputers and workstations, as well as the progress in the quickly advancing field of computer graphics.

Nowadays visualization is especially needed to make sense of ever increasing amounts of data. This increase is commonly referred to as information explosion: As technology progresses, more data is produced — more than ever before in human history. While much of this data is related to Internet user activities, the size of scientific data sets has also increased significantly. In fact, the amount of scientific data grows exponentially [Szalay and Gray, 2006]. One example is that supercomputers nowadays produce many petabytes of simulation data in fields like plasma physics and climate science [Bethel et al., 2013]. These datasets are so large that to gain insights, researchers often also need supercomputers for analysis and visualization. Less extreme examples include data obtained by modern scanning devices that are capable of producing scientific datasets in the order of

many gigabytes or even terabytes, which are often too large to be examined interactively on an ordinary desktop computer in adequate time and resolution.

As researchers want to examine objects of their study in detail, techniques have become popular that allow creating increasingly detailed models of the interior or exterior of an object, without compromising or destroying it. Modern X-ray microtomography (μCT) devices can usually produce volume models (e.g., Figure 1.1(a)) of objects such as biological or paleontological samples and archeological artifacts in resolutions of a few micrometers per point (voxel) [Dudak et al., 2016]. Synchrotron X-ray microtomography can even achieve resolutions



(a) (b)

Figure 1.1: *Examples of high-resolution datasets:* (a) *μCT scan of a garlic bulb (≈ 8.6 billion voxels),* (b) *Laser scan of Michelangelo's sculpture of St. Matthew (≈ 186.8 million vertices). Both images were rendered at 1140×2560 pixels display resolution and interactive frame rates, using the setup described in Chapter 2.*

of about one micrometer [Seo et al., 2012]. Lidar scans, such as those produced by the Swiss Geographical Survey, can lead to digital elevation models (DEMs) of

large landscapes with $2\,m$ resolution and accompanying images, i.e., textures, of these landscapes with resolutions well below $1\,m$. Laser scanning further provides us with surface models of large architectural or artistic objects with a resolution in the millimeter or sub-millimeter range, such as the cultural heritage models obtained by the Digital Michelangelo Project [Levoy et al., 2000]; see Figure 1.1(b).

The common metaphor for visualizing these and other kinds of data is the *visualization pipeline* [Haber and McNabb, 1990]. Such a pipeline essentially consists of a source of data and a sequence of transformation steps that are applied to that data in order to generate an image. These steps are *filters* in the broadest sense [Moreland, 2013], e.g., operations that determine which and how part of a dataset should be displayed. For volume data this can also include filters in the more specific sense of convolution operations, as known from image processing. The latter are useful components in a visualization pipeline, as they allow for tasks such as feature detection or noise reduction. A typical visualization pipeline is depicted in Figure 1.2: Raw data, e.g., a point cloud from a laser scanner, is first filtered (e.g., simplified). It is subsequently mapped, for example, into a surface model consisting of polygons, which is finally rendered. In-between data acquisition and rendering, several steps of selection, filtering, and mapping of data can be placed. This work, however, mostly focuses on the end of the pipeline, which is rendering.

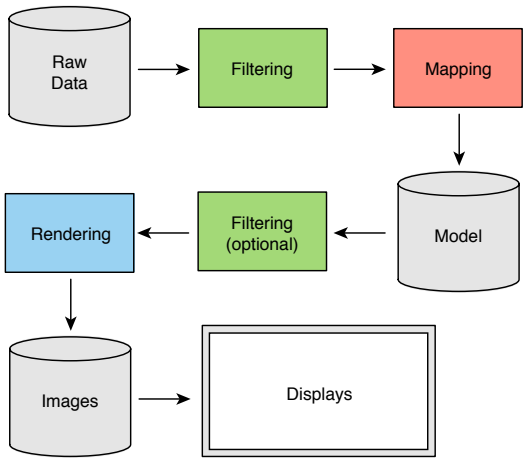


Figure 1.2: *Example of a visualization pipeline.*

1.2 Challenges

The large size of such scanning data in the range of several gigabytes up to terabytes and of even larger datasets, e.g., obtained from scientific simulation, poses

significant challenges to visualization; especially as it can be expected that the size of such datasets will only grow in future. In the domain of High Performance Computing (HPC), supercomputers with several petaFLOPS (10^{15} floating point operations per second) of computing power often also produce petabytes of simulation data — orders of magnitude larger than the aforementioned scanning datasets. Visualizing and analyzing petabytes has naturally different requirements than dealing with terabytes or gigabytes. For the visualization of large datasets, therefore, different kinds of setups have emerged, largely dependent on the amount and also the kind of data to be processed.

The larger a dataset, the more bandwidth limitations become an issue. There is a common trend that memory bandwidth tends not to grow at the same pace as computing power, known as the memory wall [Wulf and McKee, 1994]. Bandwidths of different system components also tend to not increase at the same speed. For example, network bandwidth has grown faster than the bandwidth of hard drives [Gebara et al., 2015]. Storing very large datasets on hard drive is therefore a natural bottleneck that will likely worsen in future. A solution that is often employed on HPC clusters for datasets that are too large to be stored, is known as in-situ visualization: Instead of first storing the result of a simulation on disk and then visualizing it, visualization is directly performed on the compute nodes running the simulation. This is typically done on the CPU, which produces the data and therefore has direct access to it. However, more and more supercomputers also employ GPUs which are often better suited for the task of visualization and can be used for general purpose computations (GPGPU) as well.

While in-situ rendering eliminates the data storage bottleneck, it also drastically reduces the amount of examinable data. If storage of data is therefore feasible, after simulation and sometimes an additional selection and simplification step, it is typically handed over to a dedicated GPU-based visualization system. In simple cases, i.e., if the dataset is sufficiently small or has been significantly reduced in size, this can be just a powerful graphics workstation. Often, however, the visualization system is a cluster of GPU-based render nodes.

However, not only supercomputers benefit from such nodes. HPC systems are not only usually very expensive but also exhibit power requirements and maintenance costs that are often difficult to justify. Small to medium-sized clusters of PCs, consisting of well-below 50 nodes equipped with consumer-grade GPUs and other off-the-shelf components, have become increasingly popular in recent years. Such systems are very cost effective, as they are assembled from comparably cheap components, are cheap to maintain and do not require tremendous amounts of power, due to the small amount of nodes and processors compared to an HPC cluster. These clusters of PCs are very popular and have proven valuable in visualizing gigascale and terascale datasets in a cost-effective manner. A simplified example network diagram of such a cluster-based system is depicted

in Figure 1.3 (the topic of such cluster systems is treated in more depth in Section 2.2.2).

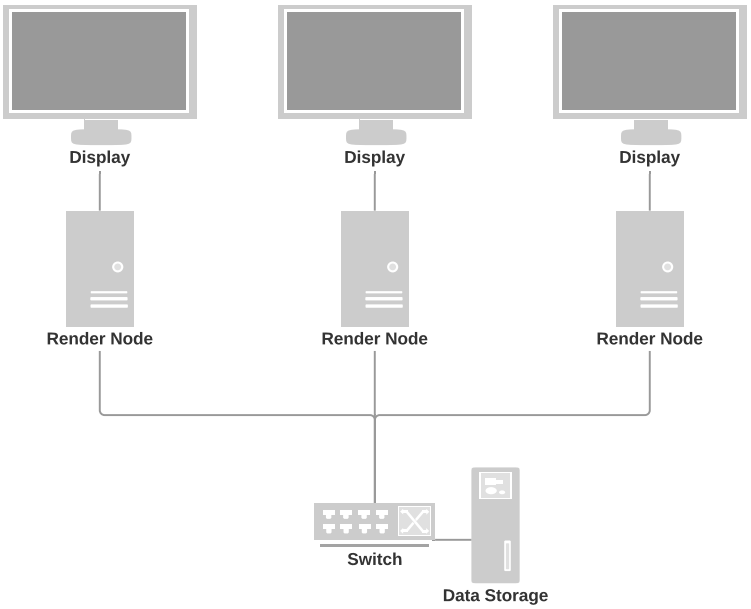


Figure 1.3: *Example network diagram of a GPU visualization cluster.*

The different configurations of visualization systems in general can also be understood as different implementations of the visualization pipeline (Figure 1.4).

Aforementioned challenges that these systems face increase further, when interactive data exploration is desired. This often means that many gigabytes of data have to be processed in order to generate a single image — and that within milliseconds. Depending on the dataset, the bandwidth and processing requirements to an appropriate visualization system can be tremendous. This is even more the case, when not only the data resolution but also the resolution of the target device is very high, as is the case for multi-projector or tiled display environments, such as CAVE [Cruz-Neira et al., 1992; DeFanti et al., 2011] or CAVE2 [Febretti et al., 2013] installations (Figure 1.5); similarly, if there are high demands on a system's frame rate, that is, the temporal resolution. Such and similar setups can be used by researchers to collaboratively (Figure 1.5b) explore massive datasets in adequate resolution, such as complex data from a neuronal simulation of a cortical column by the Blue Brain Project (Figure 1.5a). The specific challenges in creating such scalable interactive visualizations of large datasets this thesis is concerned with, are described in the following paragraphs.

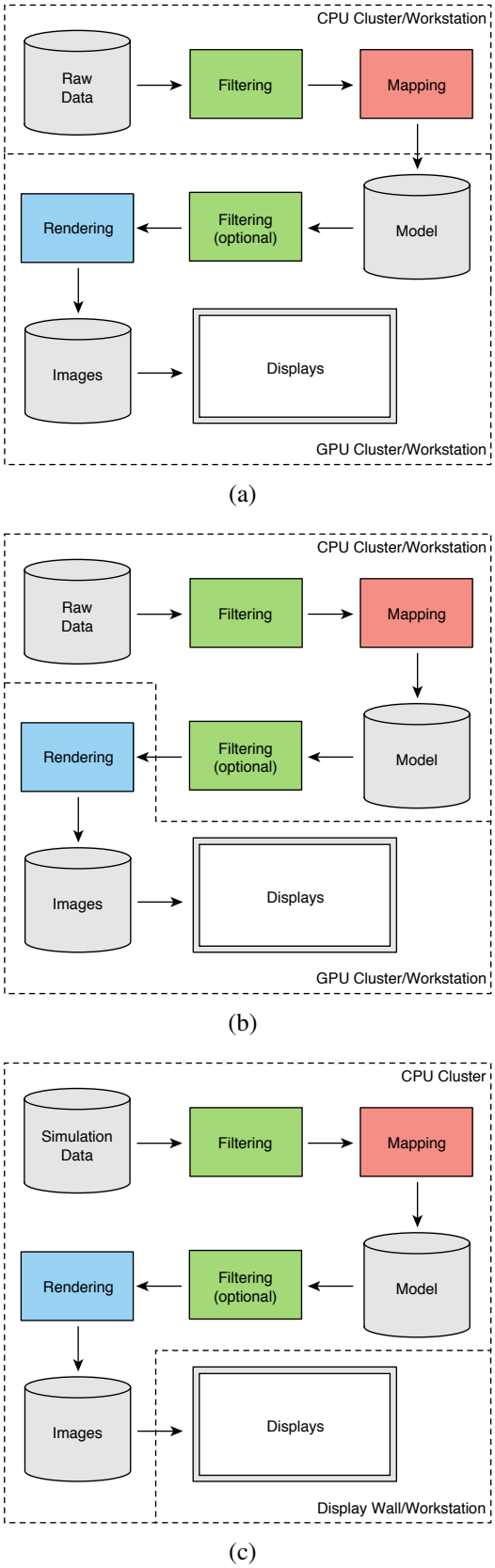
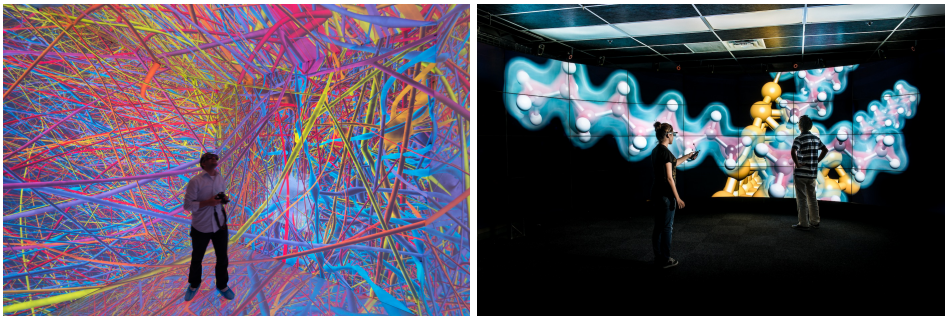


Figure 1.4: *The visualization pipeline exemplified by different visualization scenarios.*



(a) (b)

Figure 1.5: *High-resolution data visualization:* (a) 192 *megapixel CAVE installation at KAUST, visualizing neuronal simulation data with the RTNeuron software (image courtesy of S. Eilemann);* (b) 72 *megapixel CAVE2, displaying a molecular visualization (image courtesy of J.D. Pirtle). Both systems are based on the Equalizer parallel rendering framework.*

Parallel Rendering

To cope with such visualization tasks, a single GPU is typically not sufficient for rendering, as it has often neither the necessary processing power, nor the memory required to hold the data. Therefore, an additional software system is necessary to manage and coordinate several GPUs in parallel and, in fact, also several render nodes, as the possible number of GPUs within a single node is naturally limited. This approach to distribute data and rendering load among several GPUs and machines is known as parallel rendering which has its own set of challenges.

Load Balancing

Among these challenges is load balancing, that is, how to distribute rendering load among the nodes, especially in the context of interactivity. User interaction can create unpredictable changes of workload, e.g., by rapidly changing camera movement. However, users are not the only source of potentially unpredictable changes in load: The render nodes themselves might, for example, exhibit fluctuating resources because they are shared within a virtualized environment and often run different other tasks. Less unpredictable differences in workload can occur due to inherent imbalances in computational power between the nodes or simply due to imbalances in the rendered scene itself. The parallel rendering system has to distribute rendering load among the nodes equally, which can be done statically in simple cases. However, often the system has to address aforemen-

tioned load imbalances between individual nodes by dynamically redistributing workload between the render nodes.

Scalability

Another challenge is scaling the visualization system in accordance with data size or display resolution, or, more generally, increasing its performance by increasing its capacity. However, as Amdahl [Amdahl, 1967] pointed out, it is difficult to scale applications to several processors, as parallelizable tasks usually incur a non-parallelizable overhead. In the case of distributed applications, such as the parallel rendering systems we are interested in, we have to face additional overheads related to network communication and synchronization. These overheads must be considered when working with a parallel rendering system. The physical setup that we focus on in this thesis is a small to medium-sized visualization cluster, utilizing consumer-grade GPUs, as such systems have become popular due to their cost-effectiveness.

Interactive Volume Filtering

Visualizing large-scale datasets usually requires fast loading, caching, and rendering of data. It is often also required to process these datasets at interactive framerates. One example explored in this thesis is the interactive filtering of volume data for the purpose of, e.g., noise reduction or feature detection. Interactively filtering large amounts of volume data, i.e., in the range of gigabytes, creates additional challenges. To properly render and be able to store the volume in GPU memory, it is typically represented as a multi-resolution hierarchy of volume *bricks*. It is important to note that applying filter operations to these bricks at their proper resolution would yield incorrect results. Conversely, filtering the data at its highest resolution with subsequent downsampling and rebuilding of the associated multi-resolution hierarchy would yield correct results but is infeasible, due to the large amount of data involved. Even if the latter option was feasible, it naturally does not scale well and would defy using a multi-resolution hierarchy in the first place.

Generalizability

Many solutions have been proposed for parallel rendering on visualization clusters. However, they are often very domain-specific in their approach and few general solutions have been presented. The challenge here is to make parallel rendering more useful for visualizations of current and future massive datasets in various domains, such as medical imaging (volume data), geographic information systems (terrain height field data), and the examination of CAD models and

cultural heritage scans (polygonal data), while still handling load balancing and other related issues in a generic way. Therefore, solutions to the issues of parallel rendering and load balancing are arguably best implemented within a parallel rendering framework that allows related methods to be transparent to the application driving the visualization, as well as to be independent from the specific details of the actual physical setup. This layer of abstraction is also required to flexibly test parallel rendering and load balancing methods in varying scenarios with different applications. We address this challenge by implementing our methods within the flexible and generic parallel rendering framework Equalizer.

1.3 Contributions

The following contributions are presented as part of this thesis:

Novel implicit rendering task partitioning approach and corresponding affinity model

Load is balanced implicitly by the render nodes dynamically pulling *work packages*, i.e., units of work in screen-space or object-space, from a centralized queue. An *affinity model* determines which work packages render nodes will receive, based on current load and spatial coherence. We implemented the method within the Equalizer parallel rendering framework and compared it to other more traditional methods of load balancing.

Further extensions and optimizations of the parallel rendering framework

These include classes and functions for automated testing, benchmarking, and data collection, and general tools for alleviating development and debugging of parallel rendering applications, which have been used to produce the results presented in this work. These also include a utility for using optimized I/O, improving throughput when loading data from hard drive.

System allowing interactive filtering of large-scale multi-resolution volumes in the compression domain

Based on the methods and system presented in [Suter et al., 2013], our approach allows applying convolution filters, such as denoising operations, interactively to large volumes. Bandwidth requirements are significantly reduced by filtering in the tensor-compressed domain, while results are accurate — unlike with other approaches to filtering compressed or uncompressed multi-resolution volume data.

Scalability evaluation of load balancing methods within the Equalizer parallel rendering framework

We benchmarked and analyzed the load balancing methods within the Equalizer parallel rendering framework in terms of performance and data scalability (following the terminology by [Crockett, 1995]). We also analyzed how well these methods perform when facing fluctuations in available resources; varying computational resources were simulated per node, as would be the case for a heterogeneous system, and over time, as can be the case for virtualized environments.

For these experiments, we also developed a generic testbed, suitable for benchmarking parallel rendering applications, based on both polygonal as well as volume rendering, with comparable, consistent, and linearly scalable workload.

1.4 Dissertation Overview

Chapter 2 gives an introduction to the topic of parallel rendering. The state of the art is reviewed and the Equalizer parallel rendering framework, which forms the software basis of this work, is introduced. Furthermore, some of our additions and improvements to the framework are discussed and evaluated. Chapter 3 first gives a brief introduction to the topic of load balancing. After a review of the state of the art, Dynamic Work Packages, our implicit dynamic load balancing method, based on render nodes pulling work from a central queue, is presented. Its performance is experimentally evaluated in comparison to other common load balancing methods and the efficacy of different affinity models is discussed. Chapter 4 first introduces to the problem and notion of scalability in the context of cluster-based large-scale visualization. It continues then with an evaluation of the scalability of different load balancing methods, highlighting practical difficulties considering data scalability and performance scalability. The methods are benchmarked under conditions of varying load and simulated time- and node-dependent variance of computational resources. The experimental results are then discussed and evaluated. Chapter 5 is less concerned with rendering, but with filtering data in the visualization pipeline. Specifically, our system for the interactive filtering of large-scale volume data is presented. The system is based on the work by [Suter et al., 2013] and extends their system with the capability of filtering volumes in the tensor-compressed domain. The implementation and general architecture of our system, as well as our changes to the original architecture, are discussed. Then experimental results, comparing our approach with a more straightforward but incorrect one, are explained and discussed. Chapter 6 draws conclusions from the presented work and summarizes the results. Finally, further possible areas of research are sketched and briefly outlined.

CHAPTER 2

PARALLEL RENDERING TOOLS AND METHODS

While the previous chapter outlined the main concern of this thesis, which is scalable visualization, this chapter focuses on the means, i.e., the methods and tools, to achieve the ends described in Chapter 1. In other words, this chapter serves the function of describing *materials and methods*.

Paramount for building scalable systems is parallelization in general (see Chapter 4). For scalable visualization of large datasets specifically: (further) parallelizing the rendering process, that is, parallel rendering. It is a core concept to which the approaches described in this work are related; Section 2.1 therefore gives a brief overview of the topic. Subsequently, the practical implementation of a parallel rendering system is addressed. The methods and systems described in this thesis, as well as the tools for their evaluation, are implemented on top of, or as an extension of, a preexisting flexible parallel rendering library that is briefly described in Section 2.2.1. Other tools that are relevant for this work are described in Section 2.3. The used datasets are, conversely, described in Appendix A.

Moreover, the experimental setups that we used to perform all experiments related to parallel rendering in this work are described in Section 2.4, in terms of hard and software.

2.1 Overview

Research in parallel computing that exploits computational resources concurrently to work towards solving a single large complex problem has pushed the boundaries of the physical limitations of hardware to cope with ever growing computational problems. While reducing the workload of a single computational unit with parallelism in data or task space, making use of distributed parallel computers brings its own set of issues that need to be addressed. Among the main challenges are the stringent requirement for optimization of task partitioning as well as distribution of tasks to resources with consideration of minimal communication and I/O overheads.

With the dramatic increase of parallel computing and graphics resources via the expansion of multi-core CPUs, the increasing prevalence of many-core GPUs, and the growing deployment of clusters, scalable parallelism is well supported on the hardware level. In a number of application domains such as computational sciences the utilization of multiple or many compute units is nowadays commonplace. Also modern operating systems and desktop application programs increasingly exploit the use of multiple CPU cores to improve their performance. Moreover, GPUs are increasingly used to speed up computationally intensive general tasks.

The growing deployment of computer clusters along with the dramatic increase of parallel computing resources has also been exploited in the computer graphics domain for demanding visualization and rendering applications. Here GPUs are exploited using their data-parallel many-core architecture. The combination of cost-effective and integrated parallelism at the hardware level, as well as widely supported open source auxiliary software, has established GPU-based clusters as a commonplace infrastructure; for development of more efficient algorithms for visualization as well as for using generic platforms that provide a framework for parallelization of graphics applications.

Molnar et al. [Molnar et al., 1994] first described a taxonomy on the sorting stage in parallel rendering, based on the occurrence of the visibility sort in the rendering pipeline, as shown in Figure 2.1. Accordingly, three main categories of single-frame parallelization modes can be identified: sort-first (image-space) decomposition divides the screen space and assigns the resulting tiles to different render processes; sort-last (object-space) does a data domain decomposition of the 3D data across the rendering processes; and sort-middle redistributes parallel processed geometry to different rasterization units.

While GPUs internally optimize the sort-middle mechanism for tightly integrated and massively parallel vertex and fragment processing units, this approach is not feasible for parallelism on a higher level. In particular, driving multiple GPUs distributed across a network of a cluster does not lead to an efficient sort-

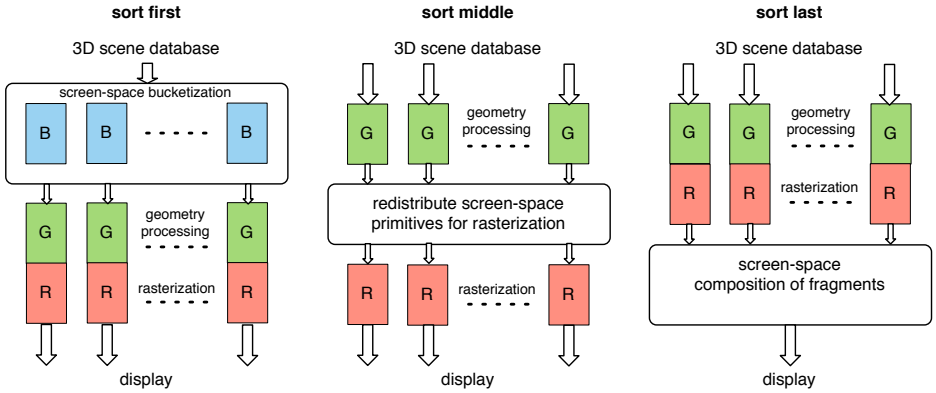


Figure 2.1: *Sort-first, sort-middle and sort-last parallel rendering workflow.*

middle solution as it would require interception and redistribution of the transformed and projected geometry (in scan-space) after primitive assembly [Samanta et al., 2000; Abraham et al., 2004].

Hence, GPUs are often treated as one unit capable of processing geometry and fragments at some fixed rate, while load balancing of multiple GPUs in a cluster system is addressed on a higher level, exploiting sort-first or sort-last parallel rendering.

2.2 Parallel Rendering Systems

Many existing approaches to distributed parallel rendering are very specific in their applicability, targeted towards domains such as geovisualization [Li et al., 1996; Johnson et al., 2006] or volume rendering [Garcia and Shen, 2002; Nie et al., 2005]. Some more generic approaches exist, mostly relying on either sort-first or sort-last methods [Samanta et al., 1999; Correa et al., 2002; Müller et al., 2006; Fogal et al., 2010]. However, few solutions have been presented that are generic enough to be applicable to a variety of visualization problems. These include the following frameworks.

VR Juggler [Bierbaum et al., 2001], its extensions Cluster Juggler [Bierbaum and Cruz-Neira, 2003], and Net Juggler [Allard et al., 2002] provide an easy abstraction of a system's underlying hardware and operating system. They are capable of driving a display wall but do not provide scalable parallel rendering functionality [Staadt et al., 2003].

Chromium [Humphreys et al., 2001], which is based on WireGL [Humphreys et al., 2000], belongs to a class of approaches that can be characterized as OpenGL intercepting libraries. These are highly transparent solutions that only require re-

placing OpenGL libraries with custom implementations. These intercept all rendering calls and forward them to appropriate target GPUs, according to different configurations of a cluster of nodes. The Chromium approach can be configured for different setups but often exhibits severe scalability bottlenecks due to streaming of calls and data to multiple nodes generally through a single node. Follow-up systems such as ClusterGL [Neal et al., 2011] try to reduce the network load primarily through compression, frame differencing and multi-casting but retain the principle structural bottlenecks.

CGLX [Doerr and Kuester, 2011] is a parallel rendering framework that is similar to VR Juggler insofar that it targets display walls, but it also provides a user-friendly mechanism to configure display setups. It is also similar to Chromium in that it provides an abstraction of certain OpenGL functions. CGLX avoids the performance issues of Chromium by maintaining an application instance on every render node and only transmitting information about user interaction, view frusta and transformation matrices. However, unlike Chromium it requires an application to be ported and recompiled, since CGLX does not provide an OpenGL proxy library. It also does not support practical means for load balancing within the system.

More generic platforms support flexible resource configurations and shield the developer from most of the complexity of the distributed and networked cluster-parallel system. The OpenGL Multipipe SDK [Jones et al., 2004; Bhaniramka et al., 2005] implements a callback layer for an effective parallelization, but only for shared memory multi-CPU/GPU systems. Equalizer [Eilemann et al., 2009] is a flexible parallel rendering library that overcomes these limitations, it is discussed in more detail in the following (Section 2.2.1). IceT [Moreland et al., 2001] represents a system for sort-last parallel rendering of large datasets on tiled displays, focusing specifically on image composition strategies. LOTUS [Cho et al., 2012], conversely, is a system which focuses on configurable virtual environments on cluster-based tiled displays.

2.2.1 Equalizer

In contrast to the other aforementioned approaches, *Equalizer* [Eilemann et al., 2009] represents a unique solution that is both oriented towards scalable parallel rendering as well as flexible task decomposition and resource configuration (see also Figure 2.2).

Equalizer can be considered the most complete solution for distributed parallel rendering [Valdetaro et al., 2014]. Similar to other approaches, like CGLX, it limits the amount of data that has to be transmitted over the network and is minimally-invasive since it does not interfere with rendering itself. However, it also supports a fully distributed architecture with network synchronization, generic distributed

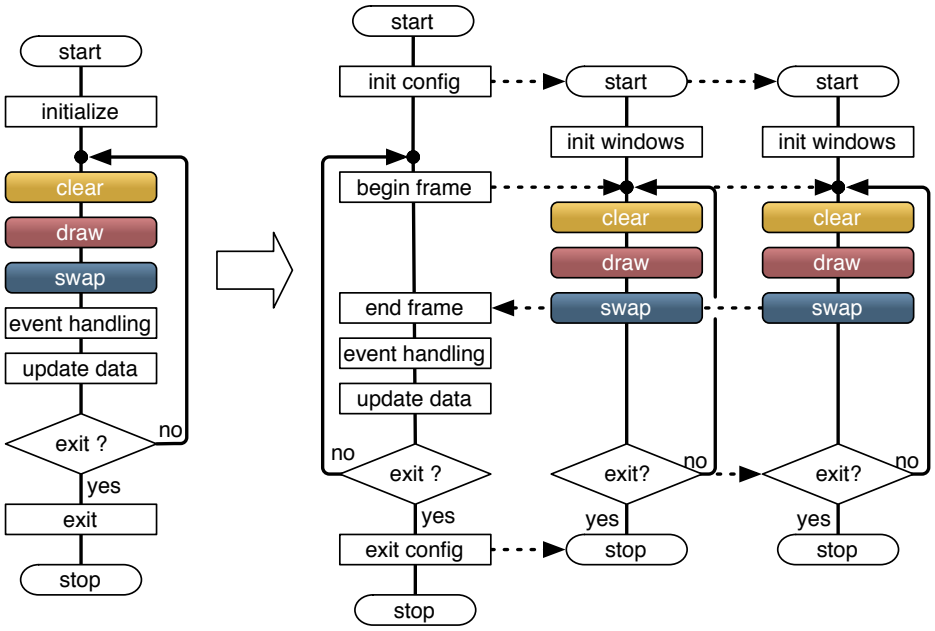


Figure 2.2: *Overview of an Equalizer server driving rendering clients based on a resource usage configuration file, as outlined by [Eilemann et al., 2009].*

objects and a large set of parallel rendering features combined with load balancing.

As a generic platform, Equalizer supports various modes of rendering task parallelization. A server configuration file declares the available resources (besides automatic detection possibilities), and allows for a flexible description of resource usage, determining the distribution of rendering tasks as well as final image composition (see also Figure 2.3). The rendering tasks can also be decomposed hierarchically into partitions in the sort-first image or sort-last data space, using a compound tree. Specific compounds are used for load balancing in 2D (sort-first) or DB (sort-last) mode, which is described in [Eilemann et al., 2009]. Such load balancing compounds are typically also called equalizers (e.g., Load equalizer, Tree equalizer).

Equalizer has also been used in the back end of Omegalib [Febretti et al., 2014], the application framework for the CAVE2™ hybrid reality environment [Reda et al., 2013], to drive a 74-megapixel, 72-tile display setup, further demonstrating relevance and scalability of the system.

Due to its flexibility and supported features, Equalizer served as a foundation for all work and experiments presented within this thesis.

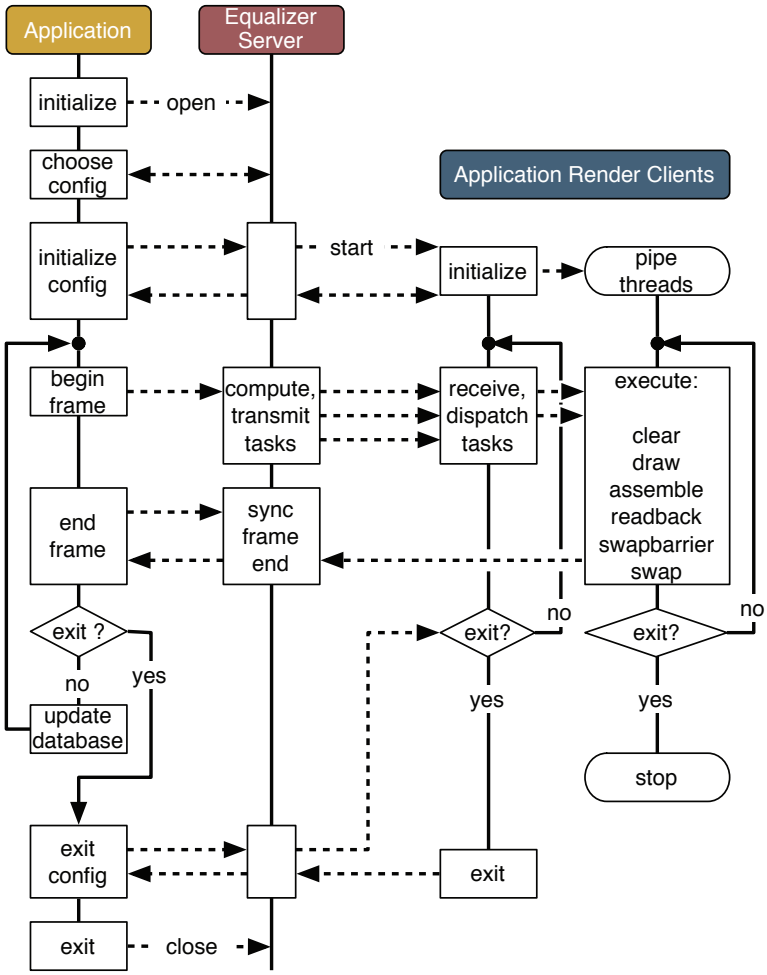


Figure 2.3: *Simplified execution flow of an Equalizer application, as outlined by [Eilemann et al., 2009].*

Abstraction Layers

From a high-level perspective, Equalizer can be understood as being based on different abstraction layers, as illustrated in Figure 2.4. The Equalizer parallel rendering library[1] itself is based on Collage[2], which implements the required distributed system and network abstraction; see also [Eilemann et al., 2018].

As "the free lunch is over" [Sutter, 2005], that is, programmers can no longer rely on Moore's law for "automatically" improving performance but instead must

[1] `https://github.com/Eyescale/Equalizer`
[2] `https://github.com/Eyescale/Collage`

turn to parallelism, another layer of abstraction is required to adequately address this issue. This role is fulfilled by Lunchbox[3], a library that abstracts platform specifics, i.e., parallelization primitives such as threads and locks, but also provides other low-level functionality, such as high performance timers and logging mechanisms which are both indispensable tools for analyzing the performance of a parallel rendering system.

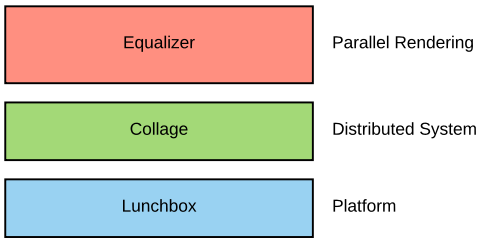


Figure 2.4: *High-level overview of the software layers that Equalizer depends on: a layer for partial platform abstraction (exemplified by Lunchbox) and a distributed system / networking layer (Collage), on top of which the parallel rendering framework is built.*

2.2.2 Cluster-parallel Rendering

The massively data-parallel nature of modern GPUs makes rendering on and effectively utilizing (making effective use of its many hundreds of stream processors) even a single GPU, symbolized by Figure 2.5(a), a parallel programming problem that must be addressed with care. A visualization system can further make use of additional compute resources (typically GPUs) within the same machine and hence *scale* (see Chapter 4 for a more detailed discussion on that topic) vertically. Such a system is typically a graphics workstation with often also multiple CPUs using symmetric multiprocessing (SMP) or, increasingly common, a non-uniform memory access (NUMA) architecture. Here each processor uses its own memory bank and therefore suffers less from contention, which is important for larger systems (Figure 2.5(b)). Although the workstation's GPUs typically need to be managed individually, e.g., by employing middleware such as the OpenGL Multipipe SDK [Jones et al., 2004; Bhaniramka et al., 2005], parallel rendering in this manner can still be performed by a single process, which can lead to a simpler software architecture.

However, if additional resources are still required because of increasing demands to throughput, for example, for visualizing larger datasets at higher frame rates or screen resolutions, the system can be scaled horizontally by utilizing additional *render nodes*. This leads to a distributed parallel rendering system that

[3] `https://github.com/Eyescale/Lunchbox`

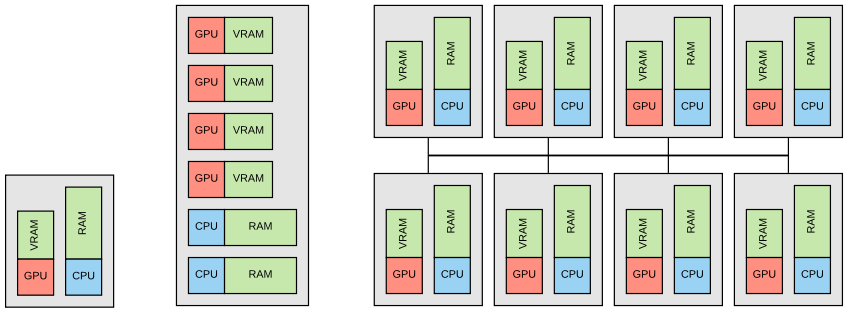


Figure 2.5: *Rendering platforms at different scales: Single machine with single GPU (a), graphics workstation (b), visualization cluster (c). Note that CPUs themselves typically employ up to ten cores (while GPUs tend to have many hundreds of cores / stream processors).*

typically also requires distributed programming models. Its nodes commonly constitute a *visualization cluster* (Figure 2.5(c)), a system used for interactive cluster-parallel rendering. Due to this intended use, it exhibits very different characteristics when compared to other systems from the domain of high performance computing (HPC). Examples are those creating off-line visualizations of complex simulations, or render farms which are used for the production of non-interactive computer-generated images for, e.g., animation films.

A system for the purpose of interactive visualization, conversely, is often constructed in the manner of a Beowulf cluster [Sterling et al., 1995], which has become a rather popular design. It refers to a small cost-effective cluster system with about a dozen nodes mostly build from commodity PC hardware and based on Linux (or another Unix-like operating system), where individual nodes usually share a common configuration and home directory. Although there are specialized Linux distributions for such use cases, like *Rocks*[4], we found it to be more practical to simply rely on a very common and well-maintained distribution (Ubuntu) instead, due to the long update cycles of the specialized distributions. In this case the cluster system can be maintained effectively via a common configuration management tool such as *Puppet*[5].

Besides being built from commodity hardware (and software), each of the nodes in such a visualization cluster also provides one or more (typically consumer-grade) GPUs, but can be connected to the cluster via a high-throughput, low la-

[4] `http://www.rocksclusters.org/`
[5] `https://puppet.com/`

tency interconnect, like Infiniband (IB). In concert, this yields a cost-effective and potentially powerful solution for large-scale interactive visualization. Naturally, such a distributed parallel rendering system requires additional middleware, in our case Equalizer [Eilemann et al., 2009], to effectively and efficiently manage resources and their usage.

Figure 2.6 illustrates example setups of such a cluster, which operate as follows. The server sets up the visualization session, based on a configuration file. In the example case, this means rendering a dataset on a small video wall with 2×2 displays.

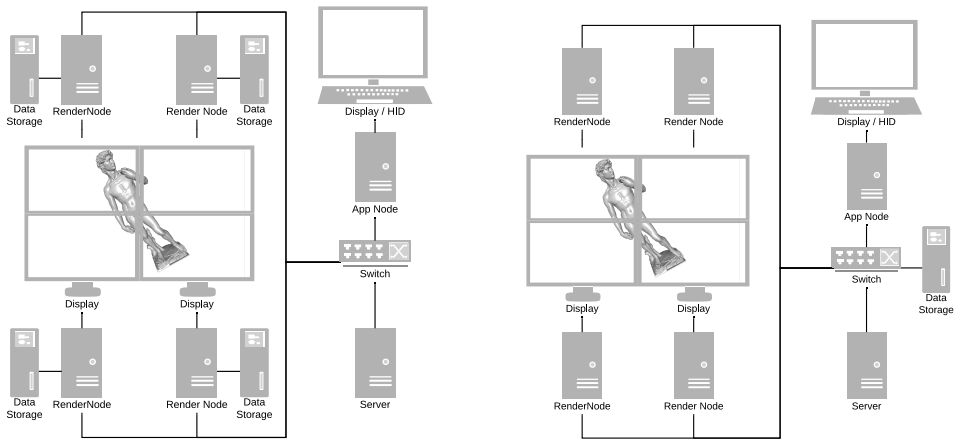


Figure 2.6: *Examples for cluster-parallel rendering setups: using local hard drives as data source (a) or using a network-attached storage (b).*

The render nodes load the data to be visualized from a data source; this can be the node's own local hard drive (Figure 2.6(a)) or a network-attached storage (NAS), as depicted in Figure 2.6(b). A centralized NAS has the advantage of being more easily maintainable, whereas using a render node's hard drive as data source requires replicating the data among all nodes and keeping each of these instances up to date. However, being at the bottom of the system's memory hierarchy, the bottleneck when loading data are naturally its hard drives. In other words, the hard drives of the NAS in (b) must be able to serve all of the render nodes simultaneously with the same throughput that each of their individual local hard drives achieves, which is unlikely for a small cost-effective setup. Moreover, such a centralized approach does not scale well with increasing system size. We consequently found replicating data among render nodes to be a more practical solution. This can be accomplished by, for example, simply modifying the script

that launches a visualization to use a utility like *rsync*[6] for efficiently updating the data file on the targeted render nodes to avoid consistency issues. This approach is naturally only viable if the visualized data changes infrequently. Although for similar use cases, there are specific cluster file systems for HPC, such as Lustre[7], these naturally incur overhead costs. Their use is consequently difficult to justify in small and simple setups.

In the shown example, based on the specified configuration, the render nodes each draw one quarter of the final image and display the result on their attached monitor, while the application node controls the visualization and also provides user input.

It is important to note that none of these roles (application node, server, render node) need to be dedicated and every node can assume multiple roles. As outlined by [Eilemann et al., 2009], an Equalizer visualization setup (and how to use resources within the cluster) can further be redefined at run time by simply changing the server configuration file. Figure 2.7 illustrates example configurations for the cluster depicted in Figure 2.6; the sort-first configuration in (a) results in each of the four nodes rendering a portion of the view frustum which is then finally displayed on the application node. The sort-last configuration in (b), conversely, leads to those nodes each rendering a subset of the object's polygonal data; the individual renderings are then combined by Equalizer via a compositing algorithm such as direct send, as described by [Eilemann and Pajarola, 2007].

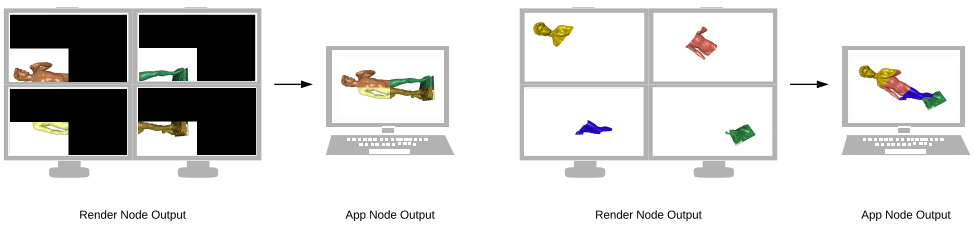


Figure 2.7: *Different configurations for the cluster depicted in Figure 2.6: Sort-first (a) and sort-last (b) configuration.*

[6] `https://rsync.samba.org/`
[7] `http://lustre.org/`

2.3 Tools and Extensions to Framework and Libraries

When developing and evaluating parallel rendering systems, especially in the context of research, certain methods and tools are required for the following purposes: to be able to extract useful information, experiments in general need to be controlled and reproducible; therefore, the process of testing the system should be automated. Furthermore, information should also be extracted in an automated fashion and the gathering of data (logging) must be reliable without affecting the performance of the system to be tested. Performance is of general concern: optimizations in this regard are not only useful for improving the overall throughput of the system, specifically optimized I/O can also speedup, i.e., improve, testing via reduced data loading times.

Apart from the tools and methods used to implement and evaluate specific parts of our systems, there is a common toolkit that applies to all approaches described within this thesis. Tools similar to those explained in the following sections are necessary to better meet the criteria listed above and needed for any serious attempt to evaluate the performance of a large-scale visualization system. In fact, many of the following tools are in part based on or inspired by similar but more limited features in the original TAMRESH system [Suter et al., 2013] (see Chapter 5). This includes more basic and less flexible functionality for screen capture, specifically for that system, as well as the possibility to create simple linear camera paths offline via manually specifying them in a text file. The following section consequently focuses on new utility functionality that we introduced within our implementation.

These are consistent with the idea of a generic parallel rendering framework and are therefore not application-specific, but covering a wide range of topics. The features described in the following have consequently been developed as toolkit and extension to the Equalizer parallel rendering framework and its foundational libraries.

2.3.1 I/O Abstraction

Equalizer's underlying library Collage provides a powerful abstraction for networking, and Lunchbox supports useful abstractions for typically platform-specific entities like threads, timers, and locks. File accesses have previously not been part of this platform abstraction. However, fast file access is typically platform-specific, and a significant bottleneck for applications dealing with massive amounts of data is limited storage bandwidth (which is the reason for techniques like in-situ visualization, as briefly discussed in Chapter 1). Of course the problem worsens as one moves down the memory hierarchy, which can make transferring giga-

bytes of a dataset through a hard disk controller, that can only support hundreds of megabytes per second, a tedious endeavor.

Such cases can be alleviated by replicating the data and reading it in parallel, as mentioned previously. Another common solution is data compression, which is briefly addressed in Chapter 5. However, a more direct approach to alleviating the issue is to also optimize data transfers from hard drive to main memory.

Therefore, we introduced a simple file abstraction to Lunchbox that allows for effective read operations, optimized for the use case of large-scale data visualization — i.e., assuming a modern out-of-core visualization system, reading large amounts of data in chunks or bricks that are randomly accessed on a hard drive and stored in an application-specific cache in main memory and/or GPU memory.

Most major operating systems provide means to fine-tune file accesses and, for example, inform the kernel about expected access patterns. Without additional information, e.g., when only file I/O functions from the C or C++ standard library are used, the kernel will typically assume a "generic" access pattern, copy pages of a file to memory when accessed, and cache some of them. The application will then read the desired file contents from memory. For an application visualizing large-scale data this is suboptimal for two related reasons. First, caching by the kernel is unnecessary, since the application typically has its own cache; i.e. the data is copied and stored twice, although only one copy is required in memory. Second, this can lead to cache pollution and slow down the system overall.

A practical solution for avoiding aforementioned issues is provided by the Linux-specific `O_DIRECT` flag for the `open` system call [Eckhardt et al., 2014] for opening a file: it allows to "bypass" the kernel and instead instructs it to copy data directly from hard drive to a user-specified buffer via direct memory access (DMA). However, this approach has the disadvantage that the amount of read data, file offsets, and the location of the target buffer in memory must be aligned with the logical block size of the used hard drive (typically 512 bytes).

Alternatively, the system call `posix_fadvise`, which is not Linux-specific, can be used to at least inform the kernel about the intended access pattern regarding an opened file. This system call has no alignment requirements.

We introduced the file abstraction class `lunchbox::InFile`, that allows for different modes when opening a file; these determine how data is read later on: `OM_DIRECT` is translated to aforementioned `O_DIRECT` flag and currently only has an effect on Linux (although similar functionality could also be implemented for other platforms), `OM_SEQUENTIAL` uses `posix_fadvise` to notify the operating system's kernel before reading from a file that all the pages within the requested range will be needed and that they will be read in sequential order, while `OM_DEFAULT` corresponds to the operating system's default behavior.

Use of this class can, e.g., look like the following:

```
lunchbox::InFile file;
if( file.open( fileName, lunchbox::OM_DIRECT ))
      file.read( offset, numBytes, dstBuffer );
else
      ...
```

Note that in this example the application itself has to take care of alignment, because of the `OM_DIRECT` flag.

Table 2.1 shows transfer rates for different modes when loading 71 bricks of the Garlic volume model (Appendix A.0.2) on a freshly booted system (to exclude caching effects): we found that the alternative file access modes can allow for significantly higher data transfer rates.

Table 2.1: *Data transfer rates for different file access modes in megabytes per second. Efficiency denotes how much of the hard drive's measured maximum transfer rate of 322 MB/s we achieved; a mode's speedup in throughput with respect to Default is listed in the right-most column.*

Mode	Transfer (MB/s)	Efficiency	Speedup
Default	202.09	0.599	–
Sequential	255.05	0.755	1.262
Direct	294.72	0.873	1.458

2.3.2 Testing and Benchmarking

When running experiments to test different methods for load balancing (Chapter 3) or interactive filtering (Chapter 5), and when examining the scalability of individual system components (Chapter 4), a reliable toolkit facilitating these tests and benchmarks in an automated fashion is essential. When evaluating parallel rendering methods, it is necessary that the load the system faces each frame stays the same across different test runs. This is illustrated in Figure 2.8: the Garlic dataset is rendered 30 times over the course of 273 frames from the same camera position. The figure shows how the rendering load increases steadily, as volume bricks are loaded (and the visible level of detail is increased), until 362 bricks are being processed and displayed. Aside from occasional outliers, the rendering load then stays very consistent until frame 139, where no rendering (and instead a guided filter operation) is performed; see Section 5.3.1 for more details about operation, dataset, and parameters. For comparing different filtering methods, as in this example, it is further important that the produced images only differ in the

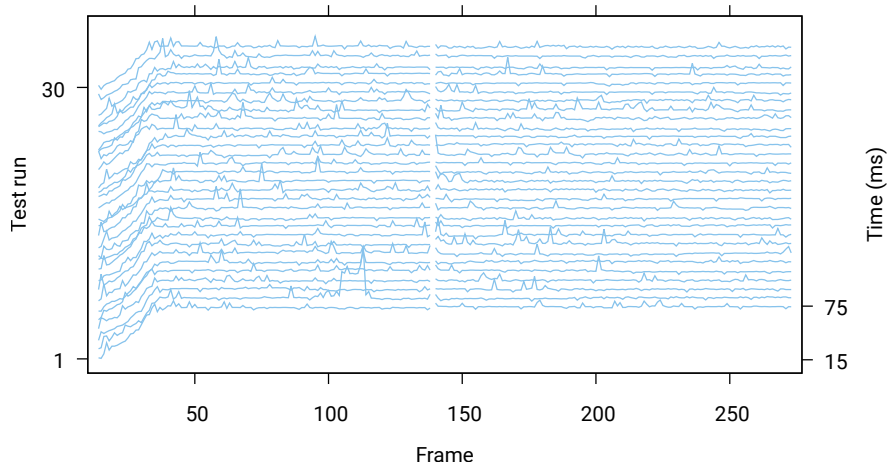


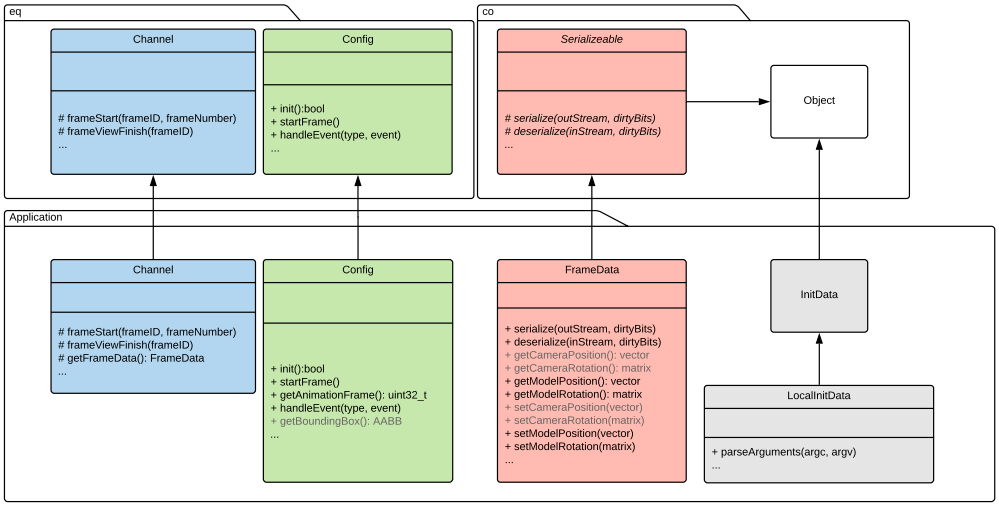
Figure 2.8: *Timings for rendering the Garlic dataset consistently* 30 *times, as described in Chapter 5.*

used method and are otherwise identical. Our approach allows us to reliably and automatically evaluate the performance of individual methods by averaging the measurements of multiple test runs.

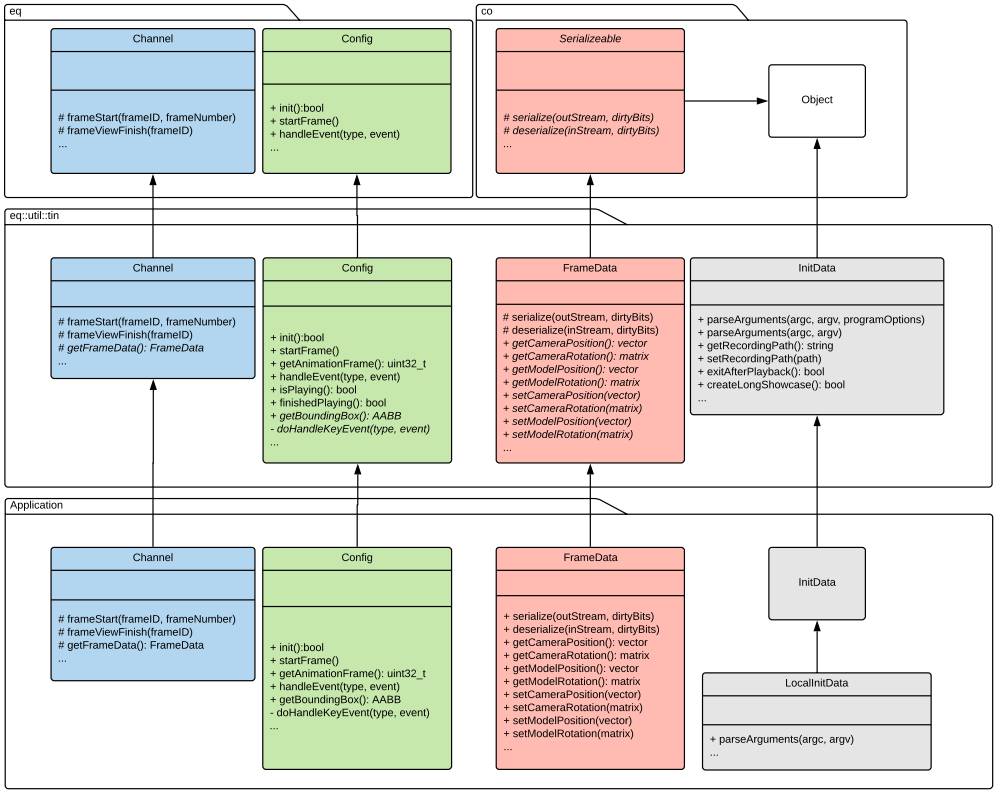
Tin

This consistency between individual test runs is ensured by a tool called Tin (as it allows us to "conserve" application test runs), which we developed as part of this work. It is an automatic testing and benchmarking framework for Equalizer applications that allows recording and replaying the state of a scene (including camera position and orientation, etc., as well as user input) on a frame-by-frame basis. Tin is comprised of a set of scripts and base classes residing in the Equalizer utilities namespace (`eq::util`). It is largely transparent to the application, which only has to adhere to a few conventions which are explained in the following.

An application can make use of Tin by deriving from custom base classes, instead of the canonical variants provided by Equalizer. This makes the application capable of recording and replaying frame data and user input, as well as capturing screenshots and image sequences; see Figure 2.9 for how this affects an Equalizer application's class hierarchy: the Tin classes essentially function as proxies for the parts of Equalizer responsible for abstracting an application's viewport (`Channel`), configuration of the visualization session (`Config`), per-frame data like model position and rotation (`FrameData`). It also provides a base class for the application's initialization data, handling information such as model paths and rendering parameters.



(a)



(b)

Figure 2.9: *Simplified class hierarchy of an Equalizer application making use of the Tin utility (b), the same application without the use of Tin (a).*

The figure also highlights the only minor required changes to use Tin: the accessors to camera data might not be typically found in an Equalizer application's `FrameData`, hence they are grayed out in Figure 2.9(a), so is the method for accessing a model's bounding volume in `Config` (that Tin requires for calculating camera paths). Conversely, implementing the private method `doHandleKeyEvent` (template method pattern) in the application's `Config` class is an interface change more specific to Tin-based applications. However, this method is only the application's renamed key event handler which is used to playback key events. In other words, these are handled first by `eq::util::tin::Config` and delegated to the application when appropriate.

Together with per-frame data, these events can then be recorded and re-played, allowing, for example, to create screenshots (typically triggered via keystrokes) or image sequences (at fixed or variable rate) at specific frames, or otherwise interact with the application. As aforementioned functionality to create screenshots or record image sequences is also provided via Tin, the developer of such an Equalizer application must further ensure to avoid collisions between the application's key bindings and those that are predefined by the Tin utility (as the latter would take precedence in this case).

Aforementioned scripts transparently allow invoking the application and subsequently *tinning* it, i.e., saving frame data and user input (keyboard), and *untinning* it, that is, replaying the recorded data and automatically storing input frame data, creating screenshots or image sequences, as well as log files together in a time-stamped folder in a directory `$TIN_BASE` (Figure 2.10). The resulting folder can in turn again be used for subsequent untinning. The figure illustrates how the scripts create a subdirectory for each application and the time stamped folders that represent individual test runs (Figure 2.10(b)). The contents of such a folder is also shown: `sequence.ccd` contains per-frame data and user input in human readable (and importantly: editable) form. Moreover, the subdirectory `frames` contains all captured frames as image files, while `stats.log` contains collected statistics for the current test run, produced by the application. The other visible files were created by a *post tin* script that is invoked for the time-stamped folder and extracts specific information from these statistics, creating a summary (`info.txt`). Besides aforementioned keyboard interaction, the user interface to the Tin utility consequently presents itself mainly in the form of script arguments, as outlined by Figure 2.10(a).

The mechanism of tinning applications allows for quick and largely application-transparent collection of data. This is achieved by simply invoking the application with a set of command-line switches for determining the target file for storing frame data, etc. These command-line switches are subsequently interpreted by the Tin class `eq::util::tin::InitData`, which the application must derive from. Similarly, it must derive from `eq::util::tin::FrameData`. These base classes

```
steiner@vranx:~$ tin
Usage: tin [OPTIONS] PROGRAM [PROGRAM_OPTIONS]
Tin PROGRAM.
        -u T,   --untin T       Untin this program with timestamp T
        -r S,   --pre-tin S     Execute script S before tinning
        -o S,   --post-tin S    Execute script S after tinning
        -d,     --delete        Delete tin after process
        -h,     --help          Display this help and exit
steiner@vranx:~$
```

(a)

```
steiner@vranx:/data/tins/volVis$ ls $TIN_BASE
eqPly  volVis
steiner@vranx:/data/tins/volVis$ ls $TIN_BASE/volVis
2018-01-22_164458  2018-05-25_214205  2018-05-31_185652
2018-01-22_164535  2018-05-26_090217  2018-06-13_184703
2018-05-18_110657  2018-05-26_090251  2018-06-13_184856
2018-05-18_110658  2018-05-31_181326  2018-06-14_145344
2018-05-18_110659  2018-05-31_181333  2018-06-14_145350
2018-05-18_110700  2018-05-31_181337  2018-06-14_145426
2018-05-18_110701  2018-05-31_181340  2018-06-14_145432
2018-05-18_110719  2018-05-31_181440  2018-06-14_145437
2018-05-18_110720  2018-05-31_181444  2018-06-14_145501
2018-05-18_110721  2018-05-31_181447  2018-06-14_145507
2018-05-18_111142  2018-05-31_182730  2018-06-20_215834
2018-05-18_111143  2018-05-31_182734  2018-06-20_215926
2018-05-18_111144  2018-05-31_182737  2018-06-20_220004
2018-05-18_111145  2018-05-31_183208  2018-06-20_220210
2018-05-24_201648  2018-05-31_184256  2018-06-20_230223
2018-05-24_201715  2018-05-31_184300  2018-06-20_230240
2018-05-25_213431  2018-05-31_184823  2018-06-20_230552
2018-05-25_213459  2018-05-31_185055  2018-06-20_231737
steiner@vranx:/data/tins/volVis$ ls $TIN_BASE/volVis/2018-01-22_164458
fps.txt  frames  info.txt  postproc.txt  reconst.txt  rendering.txt  sequence.ccd  stats.log
steiner@vranx:/data/tins/volVis$
```

(b)

Figure 2.10: *Tin usage (a), Tin directories and individual contents (b).*

merely require the application to adhere to the convention of having a dedicated data structure for storing initialization data (`InitData`) and a similar data structure for storing per-frame data (`FrameData`), such as camera parameters, which is common among Equalizer applications (such as eqPly, eVolve, etc.).

Furthermore, we integrated the Tin utilities into the major Equalizer example applications: the *eqPly* polygonal rendering application and the straightforward *eVolve* volume renderer which can both serve as a viable starting point for similar projects. We also use Tin within our system for visualizing and filtering large tensor-compressed volume datasets (see Chapter 5).

Logging

Lunchbox already provides various means of per-thread data logging via the class `lunchbox::Log`. Moreover, several macros such as `LBINFO`, `LBWARN`, `LBERROR`,

and others, defined on top of that class allow the convenient logging of data and additional debug information, based on a user-specified log level.

Based on the Lunchbox internal class `LogBuffer`, which stores log string data, we created a class more specifically tailored to performance-critical logging: `lunchbox::PerfLogger` stores a list of events consisting of time stamps and messages, only saving them to a log file when it is destroyed. Unlike the default Lunchbox logging class implementation it therefore always guarantees to only synchronize with a log file once, typically when the application terminates. As it can not always be ensured that an application terminates orderly, we further extended the `lunchbox::Thread` class (on which Equalizer threads are based) in order to catch the operating system's `SIGSEGV` (segmentation fault), `SIGABRT` (abort), and also `SIGINT` (keyboard interrupt) signals. In such cases additional log files and debug information is produced. These features further alleviate profiling and debugging parallel rendering applications.

2.4 Experimental Setups

2.4.1 Physical Setup

For the experiments within this work, we used two physical parallel rendering platforms, each representing a different level of scale.

Hactar

Type	visualization cluster	
Node	count	10
	memory	16 GB
	hard drive	322 MB/s, 158 GB
	CPU	6-core Xeon E5-2620v3, 2.4 GHz
	GPU	nVidia GTX 970, 4 GB
Interconnect	Infiniband QDR 40 Gb/s	
Software	Ubuntu 14.04	
	Kernel 3.13	
	CMake 3.7.1	
	GCC 4.8.5	
Configuration	Puppet 3.8.4	
	BIND9 DNS	

Vranx

Type	graphics workstation	
	memory (NUMA)	32 GB (16 per CPU)
	2 CPUs	8-core Xeon E5-2650v2, 2.60 GHz
	4 GPUs	nVidia GTX 970, 4 GB
Interconnect	Intel QPI between CPUs	
Software	Ubuntu 14.04	
	Kernel 3.16	
	CMake 3.5.1	
	GCC 4.8.4	
	CUDA SDK 7.0	

2.4.2 Measurements

Measurements on the CPU were performed using aforementioned Lunchbox high performance timers, timings for events on the GPU were obtained using profiling functions provided by CUDA, such as `cudaEventElapsedTime` [NVIDIA Corporation, 2017], where possible. Data was mainly collected via our Lunchbox `PerfLogger` class (or with the default `Log` class when performance was less critical), in concert with the Tin utility, as described earlier.

2.5 Summary

This chapter explained the basic principles of parallel rendering systems. The characteristics of available software systems (parallel rendering frameworks), and Equalizer as choice of such a system was outlined. We also discussed the nature of the underlying visualization clusters and the practical implementation of parallel rendering systems.

This also includes tools for high performance file I/O abstraction, and more reliable and reproducible testing and benchmarking of such systems, as well as for improved logging, debugging, and profiling.

CHAPTER 3

LOAD BALANCING

While Chapter 2 gave a brief introduction to parallel rendering systems, specific challenges in the design of such systems were only briefly mentioned and have not yet been addressed within this thesis. A major challenge among these is dynamically balancing workload within the system, especially when facing user input at interactive frame rates. That will be the focus of this chapter, which is based on our publication [Steiner et al., 2016].

Like other cluster computing systems, parallel graphics systems face the need to improve efficiency in data access and communication to other cluster nodes, while achieving optimal parallelism through a most favorable partitioning and assignment of rendering tasks to available resources. Parallel rendering adopts approaches to job scheduling similar to the distributed computing domain, and adapts them to perform a well-balanced partitioning and scheduling of workload under the conditions governed by the graphics rendering pipeline and specific graphics algorithms. Whereas some applications can be parallelized more easily with a statical a-priori distribution of tasks to the available resources, many real-time 3D graphics applications require a dynamically adapted scheduling mechanism to compensate for varying rendering workloads on different resources for fair utilization and better performance.

Distributing work to multiple resources can improve the performance of an application in general, however, the relationship between the number of resources and performance speed-up is rarely linear. As Amdahl has recognized [Amdahl, 1967], an application always contains some limiting sequential non-parallelizable

part and overhead code (see also Chapter 4), for synchronization and setting up the parallel tasks. Furthermore, the work between individual render nodes needs to be balanced for optimal speedup, which is usually difficult for real-time graphics applications. The cost of a partitioned task varies over time. For example, when a displayed 3D model is transformed on screen due to user interaction, different amounts of polygons are to be rendered for different parts of the screen. Dynamic load balancing of tasks, and assigning them to the most appropriate resources, is used to achieve a better resource utilization.

Dynamic load balancing can be defined as partitioning and scheduling the work to equalize resource utilization for better overall performance. The task of rendering an image can be partitioned within instruction or data space, i.e., into computational units of execution or subsets of data to be processed, respectively. Moreover, parameters like dependencies between tasks, priorities, and data locality should be observed while designing a load balancing algorithm. Additionally, computing the task decomposition itself should not demand a lot of resources, since it typically is a sequential portion of the code as per Amdahl's classification.

Various approaches to assign and load balance tasks for multiple resources have been proposed. In the following, we will focus primarily on interactive cluster-parallel rendering and specifically on dynamic load balancing of sort-first and sort-last parallel rendering on cluster systems. In distributed parallel rendering it is important that the workload task partitioning dynamically adjusts to heterogeneous resources, I/O and communication costs, as well as varying data dependencies and rendering costs.

We can classify load balancing into *explicit* and *implicit* approaches, where explicit methods centrally compute a task decomposition up-front, before a new frame is rendered. Implicit methods, conversely, decompose the workload into task units that can dynamically be assigned to the resources during rendering, based on the work progress of the individual resources. Explicit load balancing can be reactive, based on load distribution in previous frames, or predictive, based on an application-provided cost function. Explicit load balancing typically assigns a single task to each resource to minimize static per-task costs. Implicit load balancing generally uses a finer granularity of many more task units than resources to minimize the load imbalance due to a fixed coarse task granularity, but doing so will impose a larger per-task overhead cost. Implicit load balancing may use central task distribution or apply distributed task stealing between resources. We therefore propose a classification of load balancing methods into *reactive explicit*, *predictive explicit*, *centralized implicit* and *distributed implicit*.

In [Samanta et al., 1999], the fundamental concepts of adaptive sort-first screen partitioning and various explicit load balancing schemes have been introduced, and experimental evidence that a single task per resource leads to the best performance has been presented. In [Samanta et al., 2000], a predictive explicit ap-

proach is used for hybrid sort-first/sort-last parallel rendering. Past-frame rendering time is proposed as a simple, yet effective cost heuristic for a reactive explicit algorithm in [Abraham et al., 2004]. Pixel-based rendering cost estimation and kd-tree screen partitioning are used in [Moloney et al., 2007] for improved predictive explicit sort-first parallel volume rendering. Similarly, per-pixel vertex and fragment processing cost estimation and adaptive screen partitioning is proposed in [Hui et al., 2009]. A reactive explicit load balancing algorithm for a multi-display visualization system was further proposed in [Erol et al., 2011].

Implicit algorithms are more commonly used for off-line raytracing compared to real-time rasterization algorithms, due to the low per-tile cost in raytracing (typically expensive operations such as culling and compositing are not required). In [Heirich and Arvo, 1998], both predictive explicit and implicit algorithms are proposed and compared, and implicit algorithms are shown to be superior for raytracing. In [Korch and Rauber, 2004], centralized and distributed implicit load balancing algorithms are compared for radiosity rendering. Centralized implicit algorithms for modern, highly parallel graphics processors are proposed in [Cederman and Tsigas, 2008].

3.1 Dynamic Load Balancing

In order to improve throughput in a parallel rendering system, usage of resources must be optimized by load balancing mechanisms. These mechanisms traditionally often distribute load in a static fashion, not adapting to any dynamic changes in the rendered scene. Especially in tiled display setups, where displays are driven by different GPUs, it is common that the rendering density strongly varies among display segments, even more in the context of user interaction.

Dynamic load balancing systems must either be able to *a priori* assess the cost of the workload as accurately as possible and decompose it as evenly as possible for explicit task partitioning, or otherwise have flexible granular work units that can dynamically be assigned to the various available resources for implicit task partitioning. In the former, accurately assessing the rendering cost of some given 3D graphics data under a given viewing and illumination configuration, as well as deriving cost-uniform work partitions is non-trivial and can be costly for real-time rendering. Hence, under the assumption of strong temporal frame-to-frame coherence, most approaches use fairly simple previous-frame rendering time statistics to approximate the expected current frame rendering cost, and correspondingly, adjust the previous rendering task decomposition explicitly before starting to render a new frame. However, our *implicit* load balancing approach does neither, allowing for adaptive balancing of workload during the rendering of a single frame,

and thus is able to adapt to variable graphics resources even once the work decomposition has been defined (see Figure 3.1).

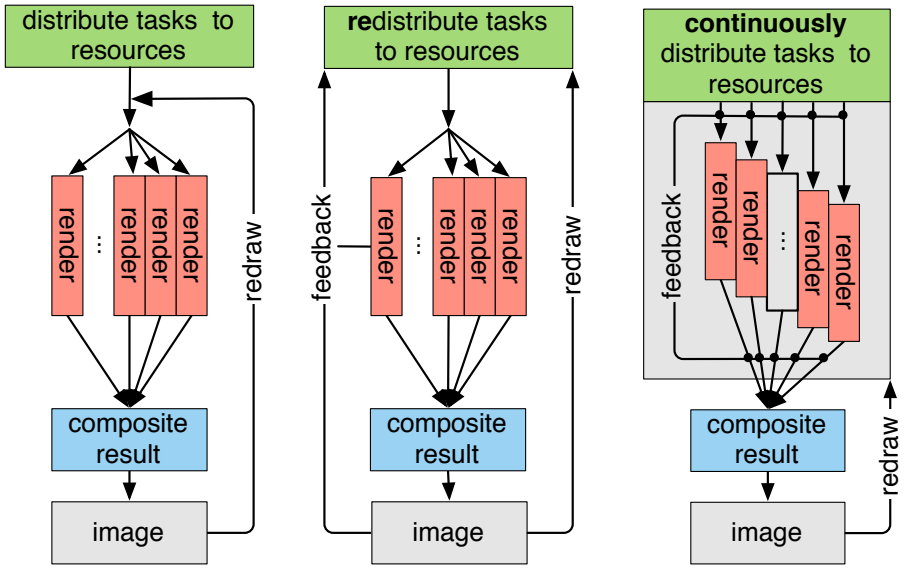


Figure 3.1: (a) *Static versus* (b) *an explicit dynamic load balancing that can adjust task decomposition between frames, and* (c) *fully adaptive implicit workload distribution.*

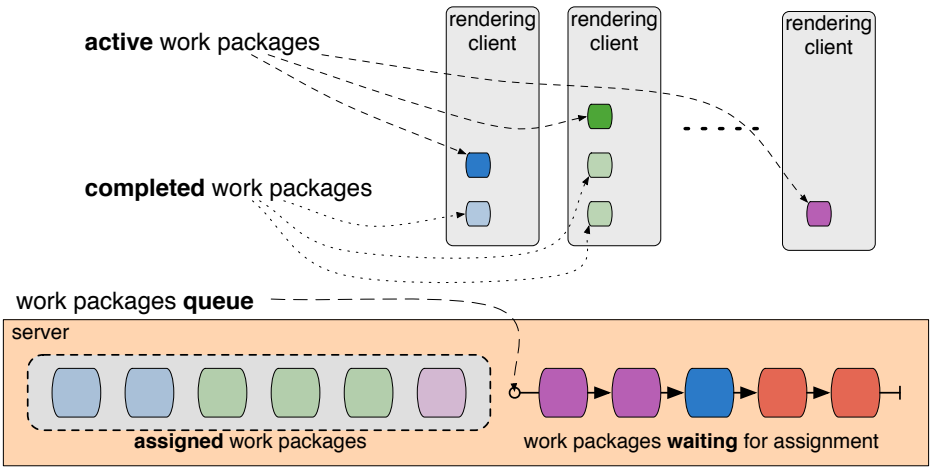


Figure 3.2: *Dynamic load balancing in distributed parallel rendering using work packages.*

Therefore, in this work we explore a flexible implicit load balancing approach (as in Figure 3.1(c)) and exploit the concept of *rendering work packages* as outlined in Figure 3.2. This allows for a quick-start setup with initial work package assignments, as well as subsequent dynamic (re)allocation of work packages to rendering resources that are ready for more work.

3.1.1 Work Package Decomposition

We implemented our work package decomposition method within the parallel rendering framework *Equalizer* [Eilemann et al., 2009]. See also Chapter 2, where this generic platform is explained in more detail. Equalizer supports various modes of rendering task parallelization; Figure 3.3 illustrates how our approach, based on work packages, is integrated within the framework.

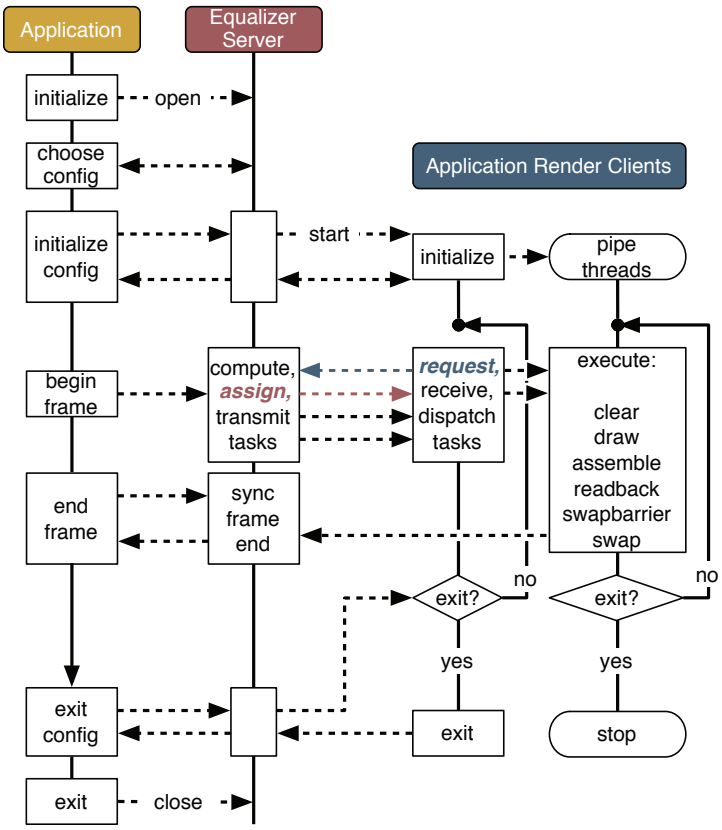


Figure 3.3: *Simplified execution flow of an Equalizer application using our work packages method. Note that clients request work packages from the server, which in turn assigns the packages to the respective client nodes, establishing a work-assignment loop that ends when all packages have been processed, finishing the current frame.*

3.1.2 Parallel Rendering Work Packages

Focusing on sort-first and sort-last parallel rendering, Equalizer already supports explicit dynamic load balancing in both image and data space by redistributing rendering tasks, based on previous frame time statistics. To further improve resource utilization, one could use a *task pulling* mechanism, an approach that has been employed before in distributed computing. We explore this approach in this work with a *dynamic work packages* implementation within the Equalizer framework. Rather than having the server push tasks to the rendering clients, our dynamic work packages approach works by managing fine grained tasks on the server side, while the clients request and execute the tasks as they become available.

As illustrated in Figure 3.2, every rendering client employs a local queue of work packages for caching purposes. During rendering, a client first works on packages from its local queue and requests n_{req} packages from the server whenever the number of available packages sinks below some n_{min}. According to the employed dynamic affinity model, the server will respond with at maximum n_{req} work packages most suitable for the requesting client. The client then adds these to its local queue.

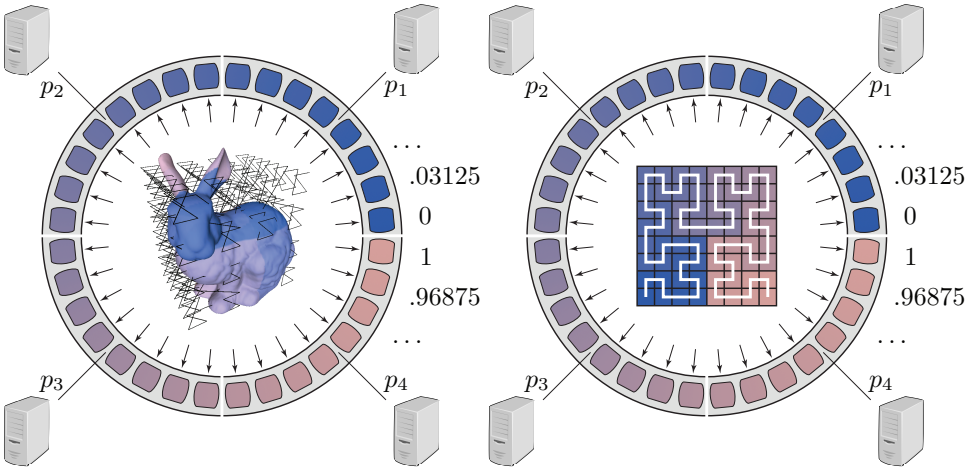
The work packages used in our system consist of small, uniformly-sized partitions in object-data or image space. At the beginning of each frame, the server generates the descriptions for all n_{total} required work packages (i.e., regions in image space or ranges in object space) and stores them in an indexed map $\mathcal{M}$. A work package is associated with, and can be retrieved from $\mathcal{M}$ with a key $k \in [0, 1]$ that is based on an *affinity* model in either image or data space, as further detailed below.

The key k is calculated from the package's index i and the total number of available packages n_{total} for the current frame as $k = \frac{i}{n_{total}}$. Given the appropriate affinity model, this corresponds to a locality-preserving mapping from data or image space to our work package key space.

3.1.3 Work Package Data Locality

To establish a data locality preserving work package affinity, we first-most must have a locality preserving linear mapping of the work packages and their data to our linear map $\mathcal{M}$ of work packages. For both, object-space data as well as image-space screen partitioning, space filling curves (SFCs) offer a locality preserving linear mapping, as illustrated in Figure 3.4. The z-curve as shown in Figure 3.4(a), e.g., can be used to map work packages of an object-space 3D data partitioning to linear indices $k \in [0, 1]$. For this, the 3D geometry data is arranged and grouped along a 3D SFC. The data locality in sort-last rendering is now achieved as fol-

lows: given that an initial data package k_0 is assigned to a certain render node, the work packages $k_{0\pm1}$ will contain spatially close geometry. Thus assigning more data packages close to k_0 to the same render node will be favorable due to less random memory accesses, and hence improves pre-fetching and caching benefits. Furthermore, nearby data work packages will be rendered to nearby regions on screen as well, thus further benefits in image compositing may be possible.



(a) 3D mapping (b) 2D mapping

Figure 3.4: *Mapping from object/image space to our one-dimensional key space using a space-filling curve in sort-last (a) and sort-first mode (b). In example (a), object-space geometry segments are mapped to work packages using a 3D z-curve. In (b), screen-space tiles are mapped to work packages using a 2D Hilbert curve. Darker colors indicate a lower, lighter colors a higher position in key space, represented as rings. The two lowest and the two highest work package positions are placed on the right-hand side. Please note that the successor of the work package at position* 1 *is at position* 0*, due to circular indexing. Four rendering clients are mapped to key space positions* $p_1 = .125$*,* $p_2 = .375$*,* $p_3 = .625$*, and* $p_4 = .875$*.*

Mapping the tiles of an image-space screen partitioning to the linear indices $k \in [0, 1]$ of a 2D SFC, together with a spatial locality preserving linearization of the 3D data, data locality in sort-first rendering can also be achieved, as indicated in Figure 3.4(b). The rendering of nearby tiles $k_{0\pm1}$ from the starting tile k_0 of a render node, will require further 3D data that is spatially close (in perspective projection) to the geometry already rendered for tile k_0. Thus locality is also preserved with respect to memory access, and further benefits may arise in the per-tile view-frustum culling stage.

3.1.4 Work Packages Affinity

For work package to node assignments, each render node is also associated with the linear key space, and given a position $p \in [0, 1]$ in this space. The work package $m(p)$ closest to this position is retrieved from the available ones in $\mathcal{M}$ according to Equation 3.2. Note that this means that the number of work packages within $\mathcal{M}$ is successively decreased over time, as more and more are requested by the render nodes. For the retrieval of work packages we use a circular addressing scheme that utilizes a distance function d as defined in Equation 3.1, which is exploited in a dynamic affinity model as further described below.

$$d(p, x) = \min\left(|1 - p + x|, |p - x|, |1 - x + p|\right) \tag{3.1}$$

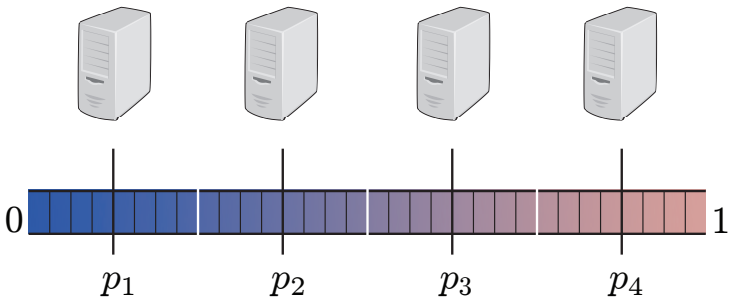
$$m(p) = \underset{x \in \mathcal{M}}{\operatorname{argmin}} \{d(p, x)\} \tag{3.2}$$

To allow the server to select the most suitable set of work packages to serve a given client request, we propose a data locality and work-load aware dynamic affinity model. As work packages are mapped to positions $k \in [0, 1]$ in our key space, requesting client nodes are associated with this space as well. In this combined work package key and node index space we define our affinity model and mapping. The key is to achieve a linear work package mapping that will eventually exploit data locality on rendering clients under a dynamic work package allocation process. The basic data locality of the work packages is achieved as explained in the previous section.

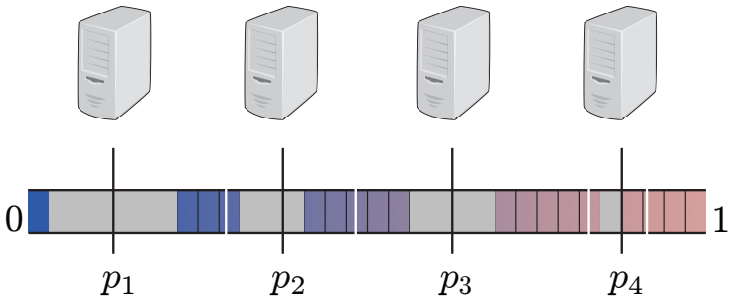
Our dynamic affinity model works such that the server maps each client to a position $p \in [0, 1]$ in key space and always responds with work packages available from $\mathcal{M}$ closest to p, according to Equation 3.2, which are subsequently removed from the map $\mathcal{M}$. Our mapping rules result in client positions and node boundaries continuously being updated as clients consume work packages, which is illustrated in Figure 3.5. Initially, clients and work packages are being mapped to key space in an equidistant fashion, as shown in Figure 3.5(a). As packages are consumed, the server continuously updates the boundaries between clients (*node boundaries*), based on the ratio of work package consumption between neighboring client nodes, as illustrated in Figure 3.5(b). Subsequently, the server re-centers client positions between adjacent node boundaries, which is shown in Figure 3.5(c). This has the effect that clients that are faster at consuming work packages will tend to move towards their slower neighbors, eventually consuming packages originally associated with these.

To preserve locality, the server only removes and assigns packages if their distance to the client's position p in key space fulfills the following condition:

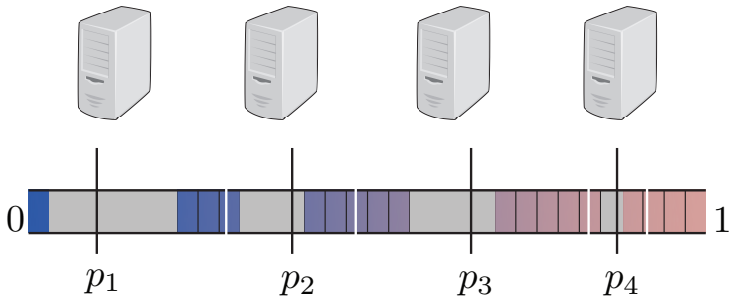
$$d(p, x) \leq d' \quad \text{with} \quad d' = \begin{cases} d(p_{prev}, x), & \text{for } x \leq p \\ d(p_{next}, x) & \text{otherwise.} \end{cases} \tag{3.3}$$



(a) Initial mapping



(b) Adjusted boundaries



(c) Re-centered client positions

Figure 3.5: *Mapping of rendering clients and work packages to key space at different stages. Showing four clients at positions $p_{1...4}$. Node boundaries are indicated by white gaps, work packages by slabs from left (position 0) to right (position 1). The color corresponding to the positions in key space. Grey areas indicate consumed packages. (a) shows the initial mapping, (b) shows re-calculated node boundaries based on client work package consumption, (c) shows client positions re-centered between node boundaries.*

where x is the position of a candidate package, and p_{prev} and p_{next} are the previous and next client's positions from p in key space, respectively. In other words: the server ensures that packages are only removed in-between neighboring nodes in key space.

To adjust for load imbalances, node positions within key space are constantly updated, according to the amount of packages they have consumed in relation to each other, within a time window w. Consequently, the number of packages used to calculate a client's position is

$$n(p) = 1 + s(p, w) \tag{3.4}$$

where $s(p, w)$ is the sum of packages the node at position p received within the last w time steps. Please note that these time steps are not dependent on frame boundaries but are currently defined as interval between two package requests being served.

The function $n(p)$ can be used to calculate boundaries between the nodes in key space as a weighted sum of neighboring node positions, based on the associated nodes' package consumption. Before serving a request, the server calculates the boundary b between a client at position p and its successor p_{next} in key space as follows, see Figure 3.5(b) for an illustration of the resulting change in node boundaries:

$$b(p, p_{next}) = \frac{n(p)p_{next} + n(p_{next})p}{n(p) + n(p_{next})} \tag{3.5}$$

The server then repositions every client in key space by centering it between the new adjacent node boundaries, as shown in Figure 3.5(c):

$$p_{new} = \frac{b(p_{prev}, p) + b(p, p_{next})}{2} \tag{3.6}$$

where p is the old client position, and p_{prev}, and p_{next} are the positions of its neighboring nodes in key space.

The role of server and client in creating and distributing work packages can be simplified and summarized as follows from Algorithm 1 and Algorithm 2, respectively. Note that their respective roles within the Equalizer parallel rendering framework are illustrated in Figure 3.3.

Algorithm 1 Role of the server (simplified)

```
1:  while running do
2:      Start frame
3:      Generate package indices and spatial positions in image or object space
4:      let n_total be the number of all available packages
5:      for each package x do
6:          k ← x.index / n_total
7:          Insert x into M at k
8:      end for
9:      Handle package requests
10: end while
```

Algorithm 2 Role of the client (simplified)

```
1:  while rendering frame do
2:      let n_local be the number of locally available packages
3:      if n_local < n_min then
4:          Request n packages
5:      end if
6:      Process server response
7:      if no more packages exist on server then
8:          Stop rendering frame
9:      end if
10:     for each local package x do
11:         Draw x
12:         Process and transmit result
13:     end for
14: end while
```

More specifically, package request handling on the server is summarized in Algorithm 3. Note that node positions are not reset every frame, but are a function of work package consumption of the respective node, taking the previous w time steps into account (see Equation 3.4), also across frame boundaries. Only before serving the very first request, this automatically results in an equidistant positioning of nodes.

Algorithm 3 Package request handling on the server

```
1:  procedure HANDLEPACKAGEREQUEST(node, n_req)
2:      Calculate boundaries between all node positions
3:      Use boundaries to recalculate node positions
4:      let p_new be the new position of node
5:      let packages be an empty list of work packages
6:      Update d'                                   ▷ see Equation 3.3
7:      package ← m(p_new)                          ▷ see Equation 3.1
8:      while n_total > 0 do
9:          if d(p_new, package.position) ≤ d' then
10:             Add package to packages
11:             Remove package from M
12:             package ← m(p_new)
13:             Update n_total
14:             if packages.count ≥ n_req then
15:                 return packages
16:             end if
17:         end if
18:     end while
19:     return packages
20: end procedure
```

3.2 Experimental Results

We tested our system on the Hactar rendering cluster (Section 2.4.1). For conducting our experiments, we used two different data sets: *David* with 56.2 M triangles and *StMatthew* with 372.8 M triangles (see Appendix A.0.2). To avoid a trivial fragment processing scenario, both data sets were rendered using a procedural marble shader and simple spherical harmonic lighting (see Figure 3.6). In all experiments, we rendered our data sets at a final resolution of 1920×1080.

In order to simulate a challenging rendering scenario, we used the Tin utility introduced earlier in Section 2.3.2 to generate a complex camera path where the camera is placed to the model very closely and moves along the major axis of the model while, simultaneously, the model rotates quickly around that axis (see also example shots in Figure 3.6). In this scenario the visibility of different parts of the model varies rapidly from frame to frame and thus the rendering load is not easily predicted by traditional load balancing mechanisms.

We implemented our method within the Equalizer framework as *Package Equalizer* and tested it with a sort-first configuration of 8×8 tiles in screen space, and a sort-last configuration of 64 segments of 3D data in object space. In Table 3.1 we

Figure 3.6: *Screenshots of the David model along the camera path. The model seems to enter the screen (right) and revolves along its longest axis while the camera moves along this axis as well (right to left image).*

compare our implicit dynamic load balancing method with conventional sort-first and sort-last dynamic load balancing approaches reliant on frame-to-frame coherence. Equalizer already contains implementations of both approaches, respectively named as *Load Equalizer 2D* and *Load Equalizer DB* [Eyescale Software GmbH, 2008].

We additionally implemented two simple affinity models for comparison to the dynamic data locality and work-load aware model that we propose. The *Equal* affinity model simply segments the key space into constant, equally-sized ranges of work packages and assigns each client to one of these for the entire duration of program execution. The *first-come, first-served* (FCFS) affinity model, conversely, simply maintains a list of work packages and assigns any requesting client with the first package available.

Our experiments are summarized in Table 3.1 which includes the draw and assembly time accumulated over all parallel nodes, and in Figure 3.7 which shows the dynamic development of draw and assembly times over time. In the latter the time reported is the passed wall-clock time for each frame, i.e., the maximum draw time needed by any node working in parallel, and subsequent assembly time. Draw time is the duration that rendering of a frame requires on a node. Assembly time is the duration required to assemble the final image, including the time to wait for all nodes to finish rendering. Increased load and load imbalances can therefore increase assembly time. The increasing assembly time as shown in Figure 3.7 is likely a consequence of the used camera path, which results in the model entering the screen from one side until covering it completely (see Figure 3.6), hence steadily increasing the assembly cost. Reaching 600 frames the model increas-

ingly covers the screen. Thus the rendering load is further increased and therefore the frame statistics plotted in Figure 3.7 only partially contribute to the timings reported in Table 3.1, which cover a longer time period. Note that the Table 3.1 summarizes *total* draw times, i.e. the sum of draw times of all nodes of all frames. Conversely, Figure 3.7 shows the effective wall-clock draw time per frame; since nodes render in parallel, this is the maximum of all draw times per frame.

Table 3.1: *Total draw and assembly time in milliseconds for the StMatthew and the David model, as well as the sum of these timings for our Package equalizer (*Pack*) and the traditional Load equalizer (*Load*) in sort-first (*2D*) and sort-last (*DB*) configurations with three different affinity models:* Equal*,* FCFS*, and* Dynamic*. The values were calculated over the duration of* 1990 *frames.*

(a) Model *David* (56.2 M triangles)

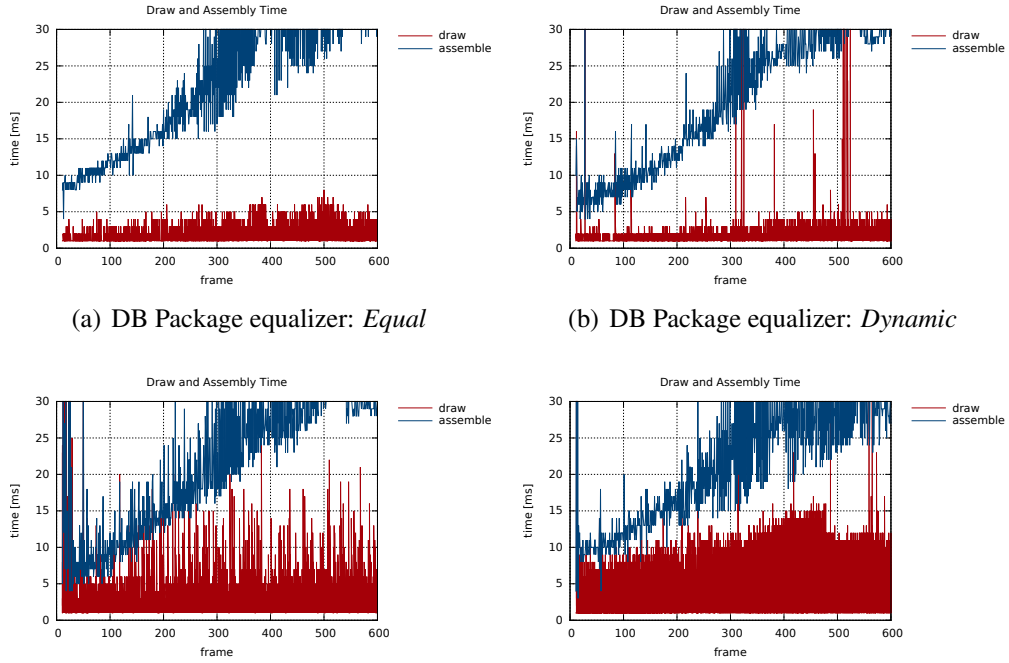
Method	**Draw**	**Assembly**	**Total**
Pack DB Equal	19940	53830	73770
Pack DB FCFS	22167	46089	68256
Pack DB Dynamic	20687	47094	**67781**
Load DB	43280	45848	89128
Pack 2D Equal	21966	10797	**32763**
Pack 2D FCFS	27464	9510	36974
Pack 2D Dynamic	23108	9985	33093
Load 2D	40034	6408	46442

(b) Model *StMatthew* (372.8 M triangles)

Method	**Draw**	**Assembly**	**Total**
Pack DB Equal	28378	65692	94070
Pack DB FCFS	114386	64270	178656
Pack DB Dynamic	31597	58404	**90001**
Load DB	93204	57136	150340
Pack 2D Equal	47193	16913	**64106**
Pack 2D FCFS	126988	26340	153328
Pack 2D Dynamic	93488	26026	119514
Load 2D	96581	9188	105769

Table 3.1 indicates that in the given sort-last (DB) parallel rendering scenarios, our method exhibits better overall performance than the traditional *Load* equalizer method, considering both draw and assembly times, as also notable in Figure 3.7.

In the sort-last scenarios, the proposed dynamic affinity model also exhibits the expected improved performance compared to the alternative *Equal* and *FCFS* models.



(a) DB Package equalizer: *Equal*

(b) DB Package equalizer: *Dynamic*

(c) DB Package equalizer: *FCFS*

(d) DB Load equalizer

Figure 3.7: *Draw and assembly times for rendering* 600 *frames of the rotating* StMatthew *model with both Package equalizer and Load equalizer in sort-last configuration (DB). The graphs result from the measurements summarized in Table 3.1.*

In the sort-first (2D) scenarios the performance of our dynamic work package method and the affinity model (*Pack 2D Dynamic*) is also better than the performance of Load equalizer for the smaller David model. However, of the tested affinity models, the simple Equal model works best in this scenario, also for the StMatthew model. This can partially be explained by the Equal model being implemented as using significantly less overhead than the Dynamic model, while, unlike the FCFS mode, still not ignoring data locality.

For the larger StMatthew model we can observe a similar behavior for sort-last (DB) rendering in Table 3.1, with the work packages method improving on the Load equalizer approach. The results indicate that unlike the traditional Load equalizer, our method seems to indeed benefit from larger problem sizes (which will be investigated in more detail in Chapter 4). For sort-first (2D) rendering, however, only the Equal work package affinity model is faster than Load equalizer.

Among the factors that affect the performance of our method are the costs and effects associated with tile-based rendering of large-scale geometry that relate to data traversal and culling.

3.3 Conclusion

We presented an implicit dynamic load balancing method for parallel rendering using a flexible rendering task partitioning approach and a novel work package pulling mechanism. In particular, we also introduced a dynamic affinity model for scoring the mapping of rendering tasks and computing resources to the same linear indexing space.

The results of our tests using the dynamic work packages method for rendering the given models using a challenging camera path in sort-last (DB) configurations, revealed a performance advantage of our method over a traditional load balancing method, based on the rendering times of previous frames.

In the tested sort-first (2D) configurations, the method for partitioning the rendering tasks in small work packages also exhibited overall better performance than a traditional load balancer. However, in this scenario the dynamic affinity model was not superior. Overhead costs and other effects of rendering, such as culling costs that grow with geometry complexity, likely contribute to this. However, this may be less the case for large-scale volume rendering where less culling overhead can be expected. Such a scenario is examined in the following Chapter 4 in more detail.

Finally, the higher performance of our method in the tested sort-last (DB) configurations, in comparison with traditional load balancing, based on previous frame rendering times, suggests that our dynamic load balancing method is highly adaptive and can react more immediately to rapid changes in the distribution of the rendering load. Our dynamic affinity model also outperforms alternative *first-come, first-served* (FCFS) and *Equal* models in these scenarios. For the sort-first (2D) setup, further investigation is needed to understand the effects of culling overhead, more accurate culling and possibly off-screen rendering. The performance of sort-first (2D) Package equalizer configurations in the context of more adequate, i.e., more challenging (especially volume) rendering scenarios is analyzed in the following Chapter 4.

Additionally, experiments with varying settings and inhomogeneous render node capacities are described in the following chapter, further revealing the potential benefits of dynamic work-package based load balancing in parallel rendering.

CHAPTER 4

SCALABILITY

Chapter 2 briefly touched upon the topic of scalability while explaining the concept of cluster-parallel rendering, while Chapter 3 focused specifically on load balancing as strategy for optimizing resource usage in parallel rendering systems. This chapter discusses notions of scalability in more detail and then focuses on our experimental scalability evaluation (Section 4.2) of the compounds and associated load balancing methods that are implemented by the Equalizer [Eilemann et al., 2009] parallel rendering framework (also discussed in Chapter 2).

4.1 Principles

One of the most important characteristics for a distributed system that faces high throughput and low latency demands, such as a cluster-parallel visualization system, is *scalability*. [Weinstock and Goodenough, 2006] differentiate between two types of scalability and give following definitions:

Scalability$_1$: "the ability to handle increased workload (without adding resources to a system)."

Scalability$_2$: "the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a system's capacity."

Examples for the first type of scalability include improving I/O (see, e.g., Section 2.3.1) or utilizing multi-resolution datasets and compression (see Section 5.2.2), and similar software-based optimizations. The second definition, conversely, fo-

cuses on repeatable strategy and consequently the possibility of *incremental* (and repeated) scaling, which will be at the focus of this chapter.

When scaling a cluster-parallel visualization system, one typically aims for maximal throughput and minimal latency while increasing the system's capability to use additional compute resources to solve increasingly complex problems (such as larger datasets, more expensive rendering algorithms, higher frame rates).

Within the system, vertical scaling strategies include integrating and utilizing additional hardware resources per node. As this is not often repeatable, a system exclusively relying on these strategies is *scalable*$_2$ only to a limited degree (notation according to [Weinstock and Goodenough, 2006]). This is different for horizontal scaling, which refers to adding and utilizing more render / compute nodes, making more extensive use of parallelization. Such an approach can be used for repeated and incremental scaling of the system and does not pose any immediate limits on system size. However, parallelization is governed by its own limits and costs that have to be considered when designing or implementing a scalable visualization system.

4.1.1 Parallelization

As there are limits to the performance of a single processor, the basis of a scalable system nowadays typically is parallelization. However, Amdahl's law famously states that the speedup S that can be achieved by using more processors to solve a problem is naturally limited by the portion of a program that is not parallelizable (as originally defined by [Amdahl, 1967]):

$$S = \frac{1}{s + \frac{p}{n}}, \tag{4.1}$$

where $s = 1 - p$ represents the serial and p the parallel portion (as a proportion of the original execution time) and n is the number of parallel processors. Consequently, the speedup is bound by the serial portion of the program and will never exceed

$$S_{limit} = \frac{1}{s}. \tag{4.2}$$

Although rendering is often classified as an embarrassingly (or *pleasingly*) parallel problem, i.e., $p = 1$, scaling a rendering application to multiple GPUs and nodes creates additional costs that limit scalability. These include overhead for communication and synchronization in general but also costs specific to parallel rendering, such as compositing partial images in a sort-last configuration.

4.1.2 Notions of Scalability

In the context of cluster-parallel rendering, two notions of scalability are of primary interest: *performance* and *data* scalability [Crockett, 1995]. The former is commonly known as *strong* scalability and is governed by Amdahl's law. It refers to the system's ability to increase performance, while the problem size stays the same. Consequently, speedup should increase proportionally with the number of render nodes. Data scalability, conversely, means that the system can handle increasingly large problem sizes. This translates into the common concept of *weak* scalability: maintaining at least near-constant execution time as the problem size increases proportionally with the number of nodes. It is based on the observation by [Gustafson, 1988] that, in practice, the serial portion s in Amdahl's law often stays constant while the parallel portion p "varies linearly with the number of processors" n. In other words, the serial overhead of a parallel application can often be compensated by solving larger problems.

Following the terminology by [Crockett, 1995], we can summarize that these notions of scalability allow us to focus on different properties of the parallel rendering system, that we are interested in scaling. While *performance scalability* describes the system's speedup that can be expected for a certain problem, *data scalability* allows us to estimate how well the system performs with increasing problem size. This therefore makes it easier to estimate whether, e.g., certain methods or setups are less suitable for small problems and also allows to focus on the overhead that is created by scaling the system to accommodate a certain problem size.

4.2 Scalability Study

This section focuses on evaluating the scalability of compounds and load balancing methods provided by the Equalizer parallel rendering framework. Please see Chapter 2 and Equalizer's original publication by [Eilemann et al., 2009] for an introduction to Equalizer and the concept of compound trees.

We conducted the following study for our article [Eilemann et al., 2018], where we also present part of the following results and where we outline the current features and scalability of Equalizer, which has matured into a general and versatile framework for parallel rendering applications since its original publication. Please see the article for additional details and also the websites of Equalizer's developers[1,2,3] for more context.

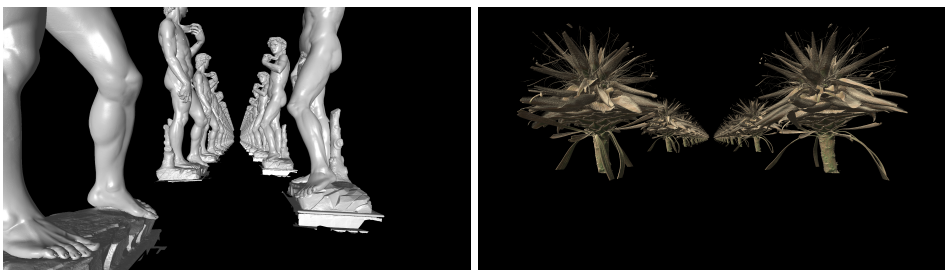
[1] `http://eyescale.ch/`
[2] `https://github.com/eyescale`
[3] `https://github.com/Eyescale/Equalizer`

4.2.1 Design of Testbed

A major challenge in evaluating the scalability of a generic parallel rendering system is designing an adequate *generic testbed*. The tests should reflect the most relevant use cases of the system (we considered the two scenarios of visualizing large-scale volume and polygonal data). Specifically, in the case of Equalizer, the tests should be equally suitable for evaluating different compounds and their associated load balancers, whereas tests using volume rendering should be comparable to those using a polygonal rendering approach. That is, both should result in similar load and the proportional change in load over time should be identical for both scenarios. Moreover, the tests must allow preferably fine-grained control over the workload within the system.

Furthermore, the testbed should facilitate insights into both performance and data scaling characteristics of the system. That is, when analyzing performance scalability, the fixed problem size must be large enough to be able to clearly observe speedup also for a larger number of render nodes. When investigating data scalability, conversely, it must be easily possible to scale the problem size (number of voxels or polygons) proportionally to the number of used render nodes.

To meet aforementioned requirements, we used the Tin utility (described in Section 2.3.2) to generate a linear camera path through an alley of objects, over the course of 800 frames. During this time span, the virtual camera is moved through the scene which consists of two rows of model instances, linearly increasing the rendering load. Each row comprises either 15 instances of the $1mm$ David model in the case of polygonal rendering (see Figure 4.1(a)), or likewise 15 instances of the 1024^3 voxel Flower model, when benchmarking volume rendering (see Figure 4.1(b)). See Appendix A for a description of datasets. The total number of 30 model instances provided a compromise between high workload and the minimum time required for running individual tests on our visualization cluster. Sort-last data decomposition is performed per scene, posing a challenging scenario for dynamic load balancing in DB mode.



(a) (b)

Figure 4.1: *Alley benchmark for polygonal rendering (a) and volume rendering (b).*

While the camera is initially placed at the origin, i.e., in the scene center with only half of the models visible, it is gradually moved backwards through the alley, until all 30 model instances are visible. This allows us to gradually increase the rendering load in a controlled manner that is consistent for both volume and polygonal rendering. Our *Alley* benchmark therefore allows us to linearly scale rendering load from

$$\begin{aligned} &\approx 8.435 \cdot 10^8 \text{ triangles} \quad / \quad 1.610 \cdot 10^{10} \text{ voxels} \quad \text{to} \\ &\approx 1.686 \cdot 10^9 \text{ triangles} \quad / \quad 3.221 \cdot 10^{10} \text{ voxels.} \end{aligned}$$

We also performed tests where we artificially created load imbalances to simulate variance in the available compute resources. We performed these only for the volume visualization scenario by varying the number of volume samples per pixel, that are used during rendering, between 1 and 7. This allows us to linearly scale workload on a per-node or per-frame basis without interfering with rendering of the scene in any other significant way, making the result more comparable to data from other tests using the same scene and camera path.

We performed tests for a selection of Equalizer compounds (based on suitability for interactive visualization) while rendering volume and polygonal data at a resolution of 2560×1440, using between 2 and 9 render nodes, as well as a dedicated server / application node, on our Hactar cluster (Section 2.4.1). This configuration with a dedicated application node that does not participate in rendering itself was chosen to ensure the render nodes being identical, as much as possible. After each test, generated performance logs were automatically collected and archived by the Tin utility and relevant data was extracted.

The test design is a compromise between aforementioned requirements for investigating performance and data scalability. For the former, to keep the fixed problem size fairly large, the task for n nodes was defined as rendering all 800 frames along the camera path, while the camera starts at the center of the scene instead of its beginning. Performance is measured as the sum of total frame durations. For evaluating data scalability, the task for n nodes was defined as rendering frames along the camera path where load is proportional to n. To measure this, the average rendering time of samples consisting of 24 consecutive frames were taken at 200 frame intervals for $n = 4$ to $n = 8$, i.e., 4 nodes render half of the scene and 8 nodes render the entire scene, while workload proportionally increases inbetween.

4.2.2 Experimental Results

On Hactar, we measured the scalability of individual Equalizer compounds, representing different load balancing methods, by rendering the Alley benchmark scene with a varying number of nodes.

Tested Compounds

We performed tests on a selection of Equalizer compounds [Eilemann et al., 2009; Eilemann et al., 2018; Steiner et al., 2016] for parallel rendering work decomposition; the compounds are briefly explained in the following list. We selected those for benchmarking, which we consider most useful for large-scale interactive visualization and also tried to focus on the better-performing compounds in each testing scenario.

Tree 2D / DB equalizer
Hierarchically subdivides either screen or object space, the resulting regions are assigned to the render nodes; splits are based on rendering timings of individual nodes.

Load 2D / DB equalizer
Similar to Tree equalizer but aims to optimize load based on a history of per-frame statistics.

Package 2D equalizer
Subdivides the screen space into tiles and performs load balancing implicitly, as render nodes themselves pull work packages (tiles) from a central queue; tile assignment is based on an affinity model. See Chapter 3 for a detailed discussion of the Package equalizer, with also more emphasis on its DB mode.

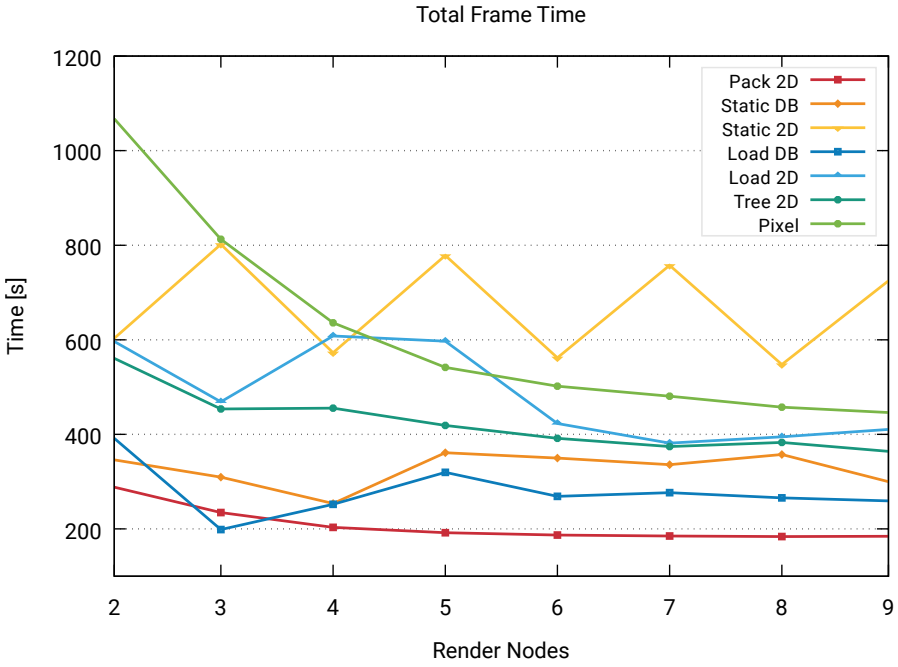
Static 2D / DB
Simple static assignment of regions in view or object space to individual render nodes.

Pixel
Each of the client nodes renders the entire scene at reduced resolution, which allows it to combine the resulting samples into a full resolution image during compositing.

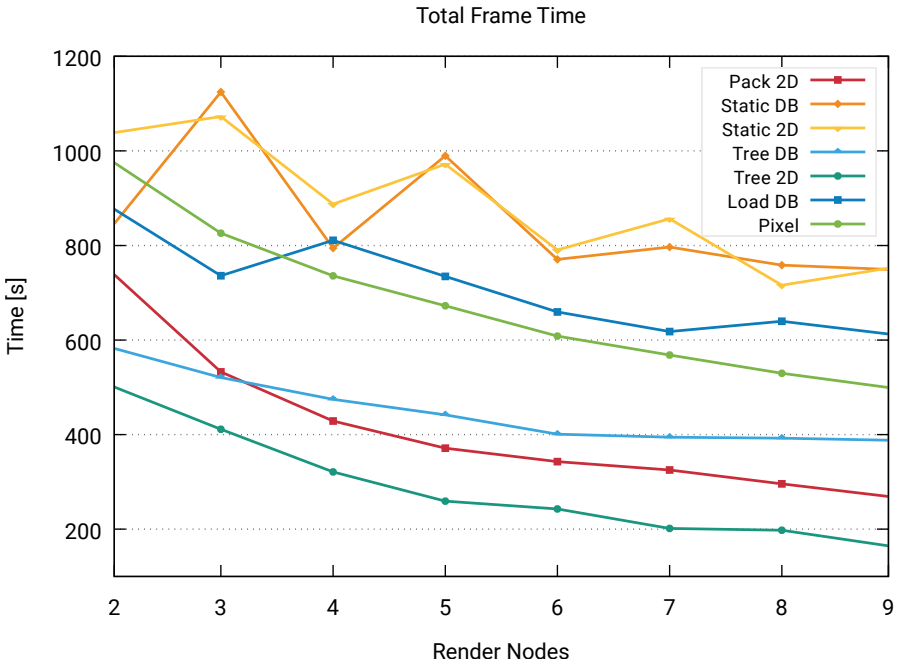
Performance Scalability

We measured the performance of individual Equalizer compounds via the Alley benchmark. The sum of rendering timings for all frames is shown in Figure 4.2 for polygonal (a) and volume (b) rendering, allowing insights into the system's performance scaling behavior.

Figure 4.2 illustrates the performance of Equalizer compounds using both static and dynamic load balancing methods. As to be expected, static load balancing performs generally worse under aforementioned dynamic load conditions. The respective Static compounds suffer from oscillations when new render nodes are added, in both sort-first (2D) and sort-last (DB) modes. This can be explained



(a)



(b)

Figure 4.2: *Timings for Alley Benchmark: total time for rendering all frames of a test scene; polygonal (a) and volume rendering (b).*

by the static decomposition generally leading to a very suboptimal distribution of workload: an unfavorable split of tiles in sort-first mode, or a suboptimal split of data ranges in sort-last mode. In the tested scenario this occurred on uneven node counts.

The Pixel compound scales with fill rates and is therefore more suited for load balancing volume rendering. However, it still exhibits a predictable scaling for both volume and polygonal rendering.

The Load equalizer compound is typically outperformed by the simpler Tree equalizer, which seemingly confirms the common notion that more straightforward systems tend to often outperform more elaborate ones in practice.

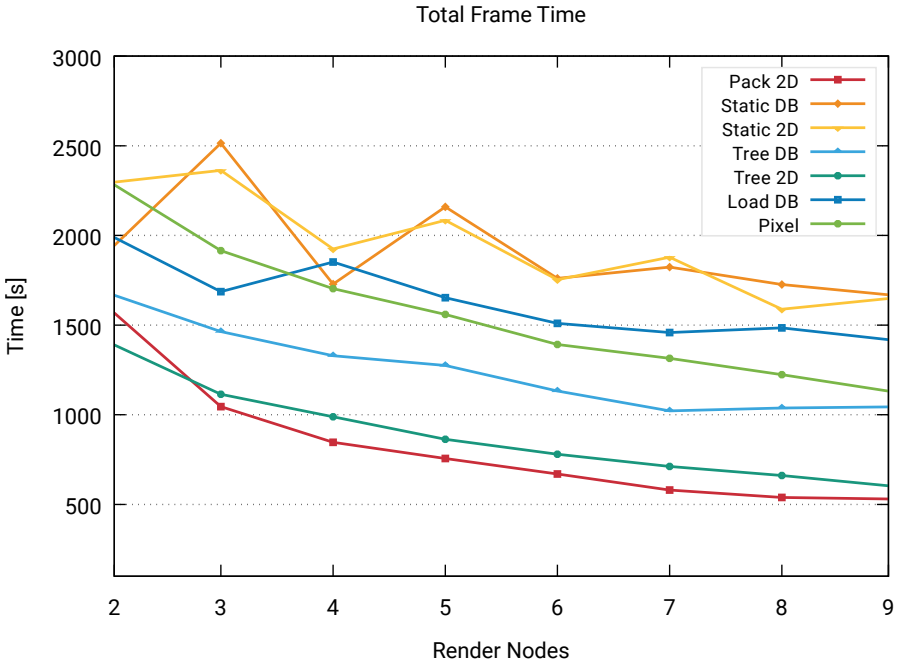
The Tree equalizer, in general, is often outperformed by the Package 2D equalizer (as exemplified by Figure 4.2(a)) and in the following we will demonstrate that this is especially true under high and variable load. The affinity model used for the Package equalizer is first-come, first-served (FCFS) using 16 packages/tiles, unless otherwise specified; see Chapter 3 for more information.

The Package equalizer seems to especially benefit from situations where load or, equivalently, render / compute resources vary strongly. This seems to again confirm the assumption introduced in Chapter 3: that load balancing methods which make fewer assumptions about frame-to-frame coherence are advantageous under such circumstances. For example, this can be the case in a virtualized environment where processes have to share graphics hardware, which can lead to strong frame-to-frame variance of compute resources. A similar case is a heterogeneous system, where resources vary per node, but where variance is typically known a-priory.

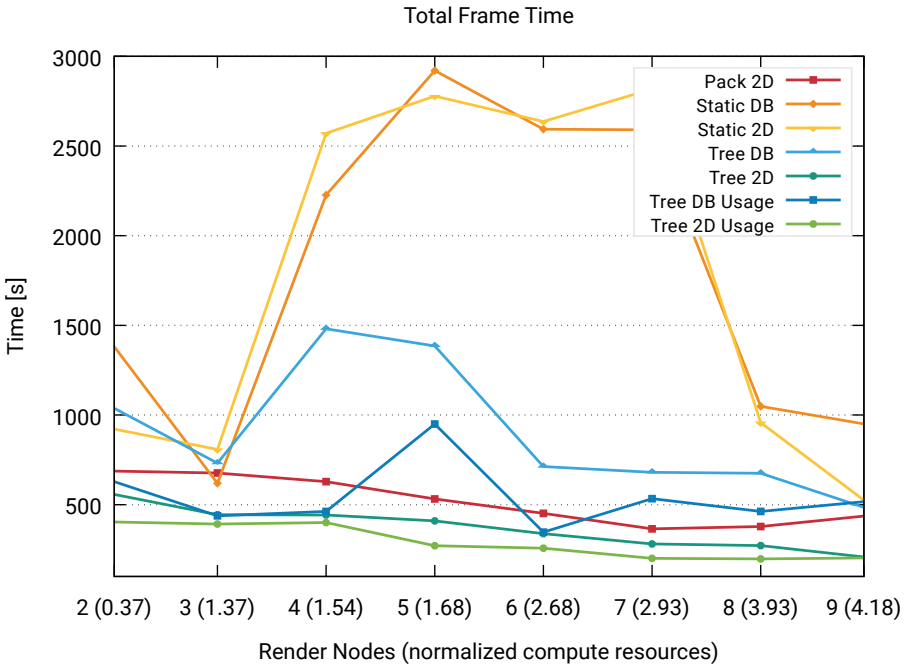
We simulated both scenarios for the volume rendering benchmark by varying the number of samples used for rendering a fragment, either per node or per frame. This allows us to linearly scale the amount of available compute resources on a per-node or per-frame basis, as shown in Figure 4.3.

For simulating a heterogeneous system, we use *normalized compute resources*, based on the idea that the availability of hardware resources is inversely proportional to their usage by the application. For example, a node rendering twice the number of samples has half the amount of compute resources available and would contribute the value 0.5 to the total number of available normalized compute resources (given in parentheses in Figure 4.3(b)).

We exploit knowledge about the availability of resources in such a heterogeneous setup and modified load balancing compounds to be capable of adjusting their *usage* of each node, i.e., a-priory bias their workload assignment scheme based on this information. We specified this usage property in the corresponding Equalizer configuration for each node individually. Figure 4.3(b) shows how the Tree equalizer benefits from that feature.



(a)



(b)

Figure 4.3: *Timings for Alley benchmark with heterogeneous load: total time for rendering all frames of a test scene; volume rendering with simulated compute resources randomly varying per frame (a) and per node (b).*

As suggested in Chapter 3, the Package 2D equalizer performs superior under load that strongly varies between frames (i.e., with available compute resources varying strongly), as its implicit load balancing mechanism makes no assumptions about frame-to-frame coherence. In the scenario where compute resources vary on a per-node basis, the Tree 2D equalizer performs best, especially when it is able to effectively utilize the available resources due to the aforementioned usage bias.

Data Scalability

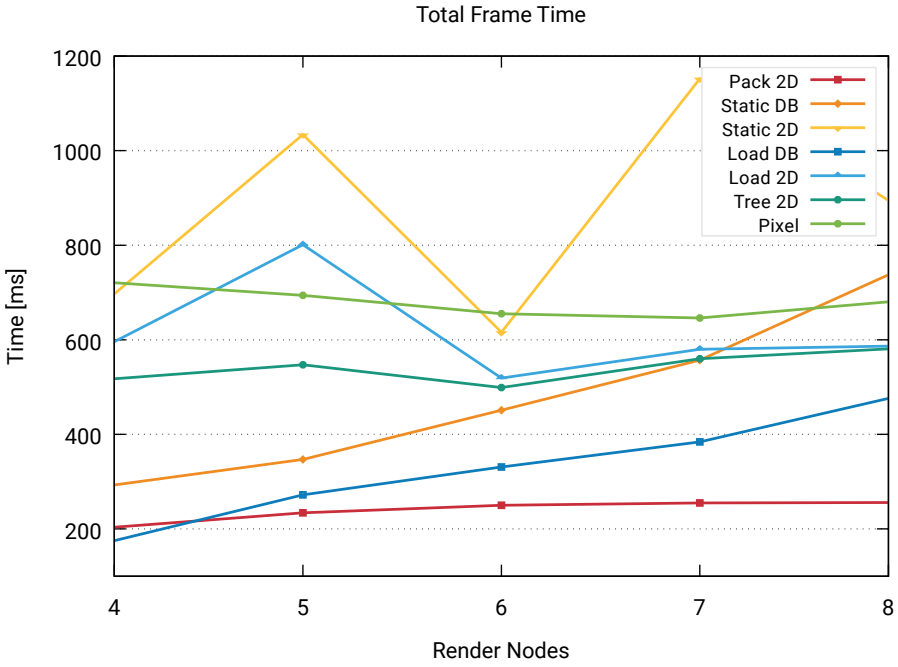
Analogous to their performance scalability, we also investigated the data scalability of Equalizer compounds. Data scalability in a parallel rendering system, as explained previously, is the system's capability to handle increasingly large datasets as its resources scale proportionally. We examined this property by rendering different proportions of the scene with a corresponding number of nodes to see how well different methods perform with increasingly large datasets; unsurprisingly, the results mostly reflect those related to performance scalability. However, as suggested above, they make certain properties of these methods more explicit and therefore complement the previously listed results.

This becomes apparent by Figure 4.4, which illustrates the data scalability of different compounds: It shows that with proportional load, for both the polygonal rendering scenario (a) and the volume visualization test case (b), the tested sort-last (DB) methods in general typically perform increasingly worse as scene size and the number of nodes increase.

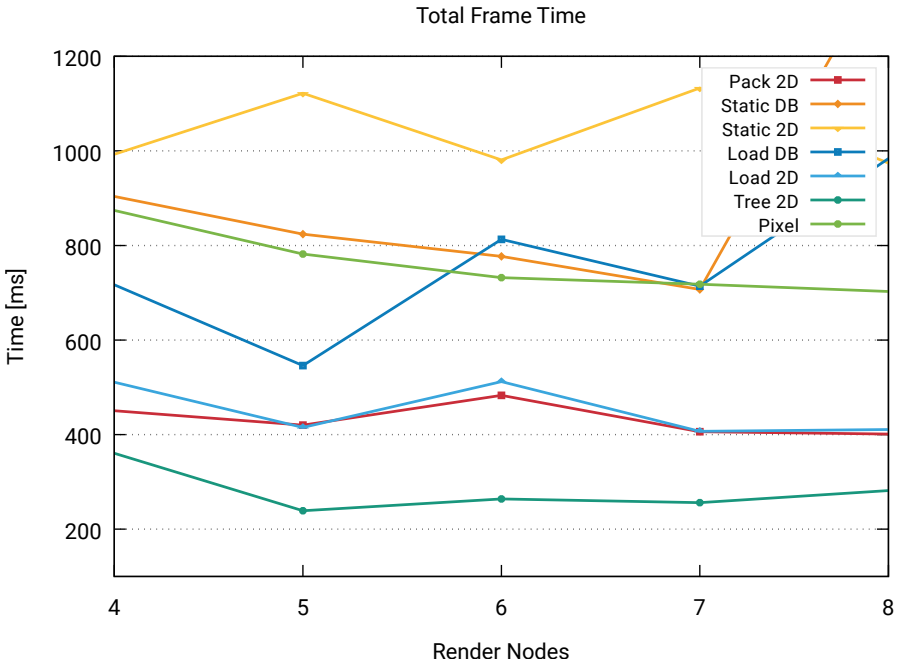
This intuitively makes more sense for volume rendering, which typically benefits less from sort-last methods and requires a more expensive compositing step (correctly alpha blending several partial images). However, for both polygonal and volume rendering this might have to do with the nature of the elongated test scene (data is decomposed per scene in sort-last mode) and resulting load imbalances and increased compositing costs.

Figure 4.4 further illustrates that, with increasing scene size and number of nodes, most sort-first compounds seem to incur mostly constant overhead for both polygonal and volume rendering. Pixel compounds, which only scale fill rates, apparently even benefit under such conditions in the case of volume rendering, with a slight proportional decrease in overhead.

The Package 2D equalizer also scales well in that regard, exhibiting a level curve for both test cases, with its data scalability being superior in the scenario where simulated compute resources vary per frame (Figure 4.5).



(a)



(b)

Figure 4.4: *Timings for Alley Benchmark with proportional load: time for rendering part of the test scene, proportional to the number of nodes; polygonal rendering (a) and volume rendering (b).*

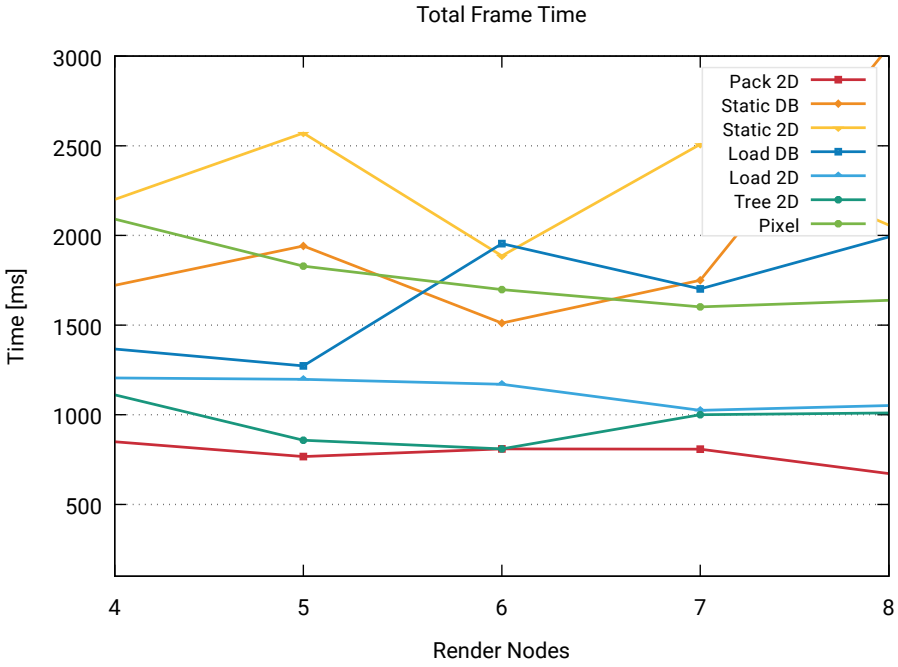
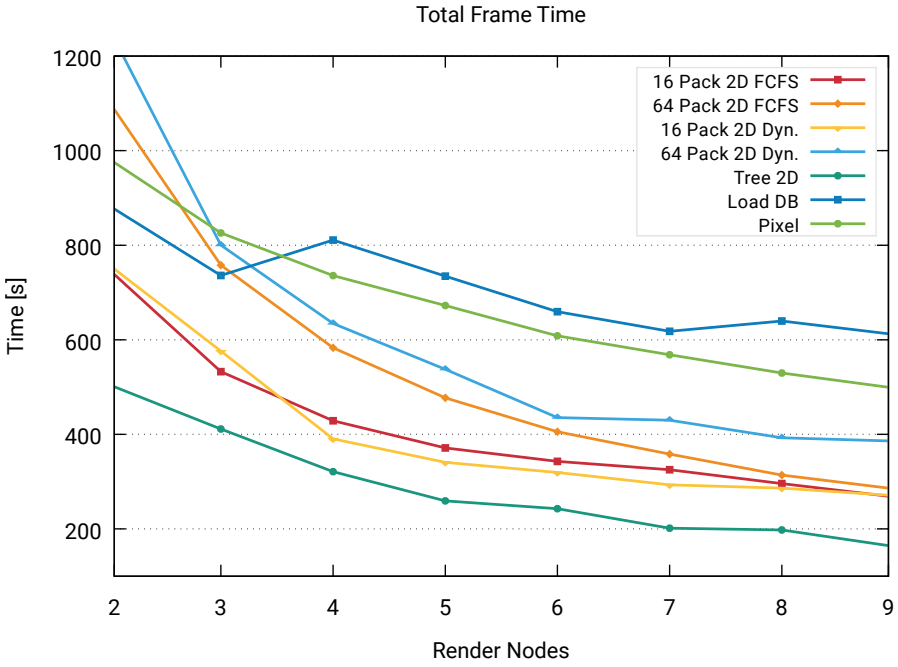
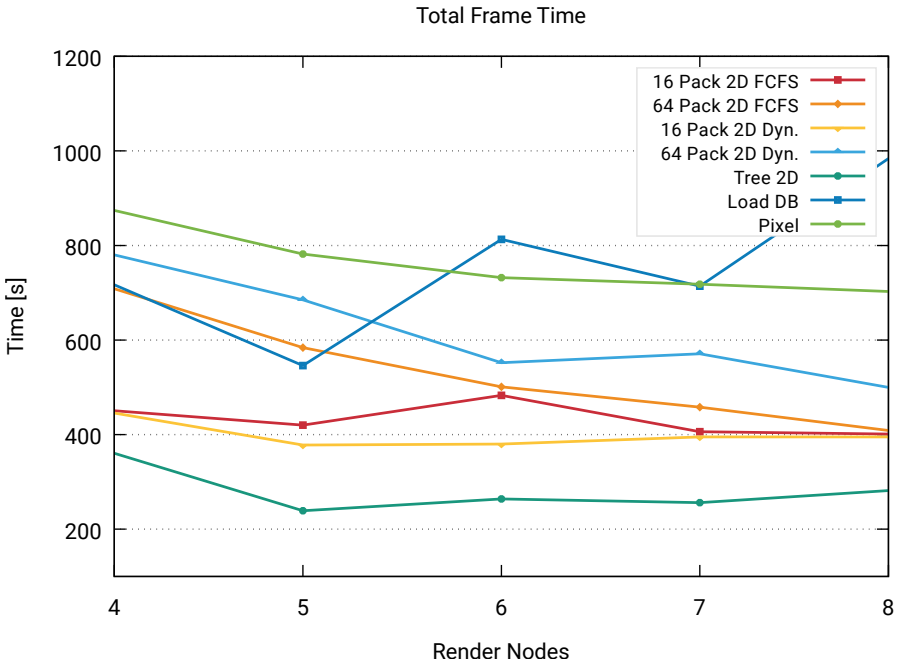


Figure 4.5: *Timings for Alley Benchmark (volume Rendering) with proportional load: time for rendering part of the test scene, proportional to the number of nodes, varying per frame analogous to Figure 4.3(a).*

As in previous figures, the affinity model used here is first-come, first served (FCFS) using 16 packages / tiles (see Chapter 3). However, for the Package equalizer we also investigated different other configurations in terms of data scalability, compared to three mostly well-scaling other compounds (Figure 4.6(b)). As to be expected, these results are mostly consistent with the corresponding performance scalability, which is illustrated in Figure 4.6(a) for comparison. Figure 4.6(b) also shows that although Tree equalizer exhibits better performance, the Package equalizers tend to show better data-scaling behavior, needing less time per frame as scene size increases. The Dynamic affinity model (see Chapter 3) introduces an additional amount of overhead that seems to depend on the number of used packages. When using 16 packages, it shows slightly better performance than the FCFS model and, unlike the latter, also a near-constant cost that suggests better load balancing for this low number of tiles. Using a higher number of packages conversely introduces a certain amount of additional cost that seems to amortize as the scene size increases (while the plot line of the Package equalizer still stays below the corresponding line of the Tree equalizer in the performed experiments). This might be due to the more fine-grained distribution of tasks to an increasing number of nodes.



(a)



(b)

Figure 4.6: *Timings for Alley Benchmark (volume rendering) with selected compounds compared against Package 2D equalizers: performance (a) and data scalability (b).*

4.3 Conclusion

We conducted a scalability study for the compounds within the Equalizer system that are most relevant for large-scale interactive visualization. We analyzed their performance and data scalability using a benchmark that allows us to compare results for both polygonal and volume rendering scenarios. Moreover, the rendering load can be finely adjusted and linearly scaled.

We found that the tested compounds seem to be more data-scalable in sort-first mode than in sort-last mode. This is less surprising in the case of volume rendering, but makes intuitively less sense in the case of polygonal rendering and might require further investigation. It can probably be partially explained by the nature of the test scenarios. Decomposition of the data is performed per scene, which also makes load balancing for the elongated test scene more difficult. Moreover, individual scene segments, if they are visible at all, typically cover a major portion of the screen, likely creating comparably high compositing costs. A sort-last decomposition per scene object might alleviate load imbalances, but yields at the same time a less challenging benchmark for dynamic load balancing compounds in DB mode. It follows that, although it is possible for benchmarks to generate finely adjustable, comparable, raw load in terms of voxels and vertices, designing a generic testbed that is equally fair for both polygonal and volume rendering applications remains challenging.

Although we analyzed both scenarios, we put an emphasis on volume rendering because it also allows us to easily scale load per fragment in a linear fashion. We used this to simulate varying compute resources on a per-node or per-frame basis, while leaving other parameters of the experiment, like camera position, unchanged. We also put an emphasis on answering the questions asked in Chapter 3, about the scalability and performance of the Package 2D equalizer compound, especially in large-scale volume rendering scenarios.

As anticipated in Chapter 3, the Package 2D equalizer compound performs well in such scenarios and, although often outperformed by the Tree 2D equalizer, shows superior performance and data scalability under conditions of strongly varying load / fluctuating compute resources.

The results suggest that, from the tested compounds, the Tree 2D and the Dynamic Package equalizer with a lower number of tiles seem to be most suitable for smaller setups. The former scales well and seems indeed suitable for most visualization scenarios. As overhead for the latter seems to increase with the number of packages, for truly massive volume visualization, the FCFS Package 2D equalizer using a larger number of work packages might be the best choice regarding the available compounds. This is based on the observation that the Package equalizer in this configuration leads to a falling data scalability curve, while the curve representing the Tree equalizer is mostly level. We hence assume

that these curves can be extrapolated. To determine whether this is truly the case, however, further experiments on larger systems with rendering problems of more massive scale need to be performed.

C H A P T E R 5

INTERACTIVE VOLUME FILTERING

Previous chapters focused mostly on the rendering aspects of the visualization pipeline. However, the visualization pipeline in its general form merely consists of transformation steps, also known as filters [Moreland, 2013], that are taken to produce a final image from a set of input data. This can also include filters in the more specific sense familiar from image processing: a class of operations which includes linear convolutions, especially useful for tasks such as feature detection or image enhancement. This chapter focuses on the integration of such convolution filters into an interactive visualization pipeline for large volume models and related challenges in terms of performance and system design.

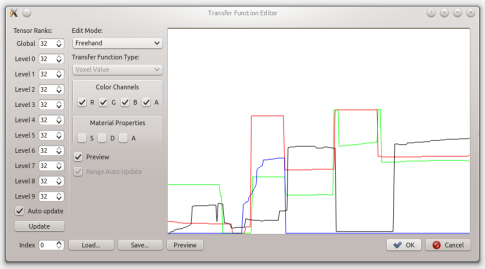
5.1 Volume Visualization

There are numerous approaches to interactive volume visualization, but increasingly powerful graphics hardware has allowed direct volume rendering (DVR) to become the most popular choice. Instead of, e.g., only performing a computationally cheaper rendering of the volume's isosurface (often extracted in a preprocessing step), DVR involves directly projecting volumetric data onto the screen, which is nowadays typically performed via raycasting. Interactive visualization of large-scale volumes also usually requires additional techniques for data compression and management. This includes multi-resolution hierarchies and similar

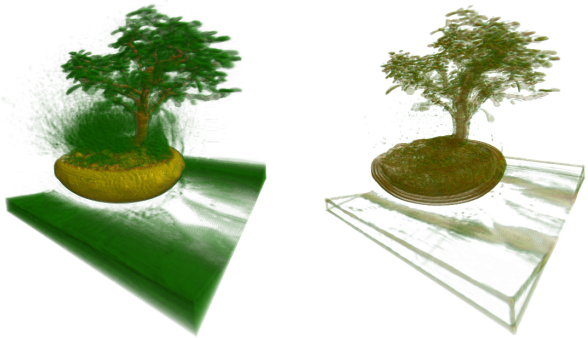
approaches to level-of-detail (LOD) management, as well as out-of-core techniques [Balsa Rodríguez et al., 2014].

As CT scan results and other volumes typically consist of density samples, aforementioned projection must be preceded by mapping optical properties, such as opacity and color, to the samples via a transfer function (TF); this is often implemented via a user-generated lookup table. Figure 5.1 shows a screenshot of the TAMRESH [Suter et al., 2013] transfer function editor in which individual color and material values within this lookup table are symbolized as curves that can be edited by the user. This allows rendering shaded and colored images of the Bonsai dataset (Appendix A.0.2), in this case. We slightly modified the editor to support editing multiple transfer functions (currently two TFs can be used by our application, before and after filtering, respectively; see Section 5.3.1).

The aforementioned mapping is referred to as classification step [Engel et al., 2006] and already provides an easy means for interactive data selection, e.g., to distinguish leaves and wooden parts of the bonsai tree in Figure 5.1(b). Trans-



(a)



(b) (c)

Figure 5.1: *TAMRESH transfer function editor (a) and resulting image of the Bonsai dataset (b); limitations of transfer functions for classification (c), highlighting of certain features such as edges is not possible with a simple TF mapping.*

fer functions further provide the possibility to create simple (point) filters, that is, without the use of convolution (for example, to adjust brightness or contrast of a dataset). However, they do not allow for more advanced operations, such as edge detection, without additional (precomputed) information about the relations of samples to their neighboring values (e.g., gradients). For example, the edges in Figure 5.1(c) were highlighted by our system with a simple difference of Gaussians (DOG) filter; this and similar operations require a visualization system to support linear convolution. It would otherwise be rather expensive in terms of bandwidth and storage requirements to provide the required additional data (relations of samples to each other) to the transfer function. This is worsened by the fact that, for example, gradients are rather sensitive to quantization and lossy data compression, as they tend to cause artifacts this way [Engel et al., 2006].

5.2 Filtering Volumes

Convolution filters are ubiquitous image processing tools; they are simple to apply and many common operations, such as edge detection or blurring can easily be expressed as a filter kernel $\mathcal{G}$. Such a kernel is a vector, a matrix, or generally, a *tensor*: a generalization of the concept of scalar, vector, and matrix, which are tensors of orders 0, 1, and 2, respectively. Consequently, a 3-dimensional kernel is a tensor of order 3, with which values in a volume dataset $\mathcal{A}$ are linearly combined around the kernel's origin, for each value in the dataset. This convolution can be expressed as

$$\mathcal{A}' = \mathcal{A} * \mathcal{G}. \tag{5.1}$$

The kernels of many popular convolution filters, such as Gaussian blur, can further be expressed as an outer product of vectors, which makes them *separable*; this means that a convolution can be performed using a set of only 1-dimensional kernels. E.g., assuming $\mathcal{G}$ is 3-dimensional and separable, Equation 5.1 can be written as:

$$\mathcal{A}' = \mathcal{A} * \boldsymbol{g}^{(1)} * \boldsymbol{g}^{(2)} * \boldsymbol{g}^{(3)}, \tag{5.2}$$

sequentially convolving the dataset $\mathcal{A}$ along its three dimensions with 1D kernels $\boldsymbol{g}^{(1\ldots3)}$, significantly reducing the computational and bandwidth cost of applying the filter, when compared to directly using a 3D kernel (Equation 5.1).

However, as complex volume datasets tend to consist of billions of voxels, applying convolution filter operations to them is still challenging. A naive approach might be to directly apply a filter kernel to all elements within the volume, which is not possible at interactive frame rates, except for very small volumes. In such a case, a filter kernel of size K^3 and volume of size I^3 with a non-separable kernel can result in $K^3 I^3$ read accesses to the volume data, whereas a separable kernel

might still require $3KI^3$ read accesses. It is easy to see that this approach is not very scalable and limited to only small volumes, if interactive results are desired.

5.2.1 Multi-resolution hierarchies

A further impediment to a naive approach to volume filtering are multi-resolution hierarchies, which are typically necessary to effectively render large volume data at adequate resolution.

To avoid aliasing, such a multi-resolution hierarchy entails storing the volume in *bricks* (3-dimensional blocks of data) of varying resolution (as downsampled versions of the original data), as illustrated in Figure 5.2. It is important to note that only those bricks are cached and selected for rendering that are needed to produce an image for the current virtual camera position.

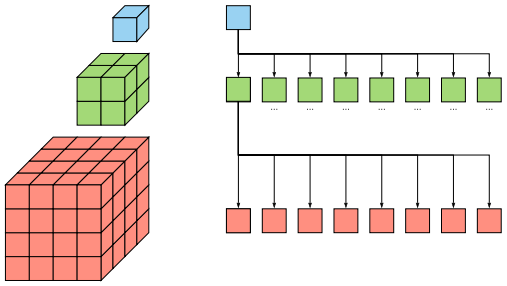


Figure 5.2: *Multi-resolution hierarchy: A volume stored at three different resolutions as a hierarchy (octree) of bricks.*

Downsampling

Note that by using the term *downsampling*, we imply that the signal has been adequately band-limited before the resampling step, according to the sampling theorem; i.e., via low-pass filtering it is ensured that the highest frequency within a signal is only half of the new sampling frequency. The term will be used this way throughout the rest of this chapter.

Although being far from the theoretically optimal bandpass, which would be the sinc function, for practical reasons it is common to use a simple box filter (which has suboptimal spectral properties) instead [Engel et al., 2006; Williams, 1983]. This is due to the fact that such multi-resolution hierarchies are typically constructed via simple successive averaging of neighboring values, which is computationally very inexpensive. For reasons of efficiency, this is also the approach that we follow throughout, in the context of downsampling.

Consequences of using multi-resolution hierarchies

In practice, using such a hierarchy for rendering means that only those bricks will be available at highest resolution, which are very close to the camera and also within the current view frustum. By only having to render and cache volume data of appropriate resolution, this common technique makes the bandwidth and memory requirements of DVR acceptable.

However, multi-resolution hierarchies pose another impediment to interactive filtering of volume data: Since most currently visible bricks are typically a downsampled, lower resolution version of the original volume data, it is not directly possible to correctly apply convolution filters to them, as the sampling rate must be consistent for both the input signal and the kernel it is convolved with. For some more robust kernels and simple operations, such as a Gaussian blur, the result can be crudely approximated by indeed downsampling the filter kernel for lower-resolution bricks (Figure 5.3).

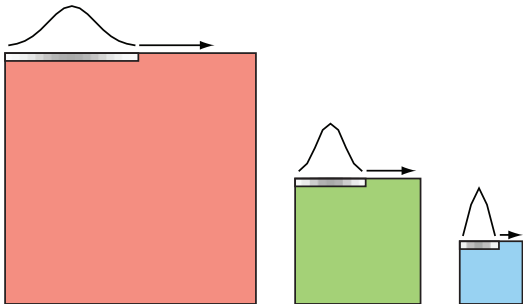


Figure 5.3: *Gaussian kernel applied to a dataset at* 3 *different resolutions: note that the kernel itself is downsampled to match the resolution of the respective level of detail.*

This naturally results in inaccurate or lost filter responses and is outright impossible for many operations. This includes cases where the filter kernel is very compact and allows no way of downsampling that preserves its essential properties. One such case is the Sobel operator. For this filter with a kernel size $K = 3$, as well as many similar operations, simple downsampling of kernels is clearly not an option. The Sobel operator's kernels $S_{x,y,z}$ calculate the partial derivatives within a scalar field, the magnitude of the resulting gradient is typically used for edge detection; 3-dimensional versions of these separable kernels are shown in Equation 5.3. In this example the convolution operator $*$ allows it to express the 3-dimensional Sobel kernel as a convolution of simpler 1-D kernels extending in x, y, and z direction, calculating gradients for each (as well as smoothing the result).

$$
\begin{aligned}
\mathcal{S}_x &= \begin{bmatrix}-1 & 0 & 1\end{bmatrix} * \begin{bmatrix}1\\2\\1\end{bmatrix} * \begin{bmatrix}1 & 2 & 1\end{bmatrix}, \\
\mathcal{S}_y &= \begin{bmatrix}1 & 2 & 1\end{bmatrix} * \begin{bmatrix}-1\\0\\1\end{bmatrix} * \begin{bmatrix}1 & 2 & 1\end{bmatrix}, \\
\mathcal{S}_z &= \begin{bmatrix}1 & 2 & 1\end{bmatrix} * \begin{bmatrix}1\\2\\1\end{bmatrix} * \begin{bmatrix}-1 & 0 & 1\end{bmatrix}.
\end{aligned} \tag{5.3}
$$

This highlights the fact that more complex convolution filters can typically be created by sequentially convolving a dataset with much simpler kernels.

Yielding correct results for non-trivial filter operations would require applying them to the version of volume data at highest (correct) resolution and subsequently rebuilding the multi-resolution hierarchy. This would not only be unfeasible for an interactive application but it would defy the purpose of using a multi-resolution hierarchy, since even bricks that only need to be displayed at a low level of detail (LOD) would still have to be processed at full resolution.

The fact that naive solutions for interactive filtering of multi-resolution volumes are either potentially incorrect (by downsampling the filter kernel) or prohibitively expensive (by filtering volume data at highest resolution and rebuilding the hierarchy) creates the need for a different approach.

5.2.2 Filtering Compressed Data

The high bandwidth and memory requirements of large volume models have led to the fact that they are commonly stored in compressed from, the data being only decompressed when required and after upload to GPU memory. Typically, effective lossy compression algorithms are used for this purpose in large volume rendering. Such methods are often based on Fourier (FT) [Lippert et al., 1997], discrete cosine (DCT) [Yeo and Liu, 1995], or wavelet (WT) [Grosso et al., 1996] transforms, while tensor approximation (TA), sparse coding, or vector quantization are further suitable techniques [Balsa Rodríguez et al., 2014]. Such compression introduces additional challenges to interactive filtering of volume data: Filtering after decompression introduces additional errors, e.g., by highlighting compression artifacts,

whereas filtering before compression would defy the purpose of using compression in an interactive setup, as volume compression is typically a very expensive operation performed off-line.

5.2.3 Filtering in the Tensor-compressed Domain

A solution to aforementioned challenges of filtering multi-resolution and compressed data is accurately performing the convolution within the compressed domain. This must entail inexpensively (in terms of computation and data access) filtering volume data at its highest (correct) resolution without introducing additional errors, while also allowing to quickly rebuild the required multi-resolution hierarchy of volume bricks. Suitable methods can be based on tensor approximation (TA), which has proven to be effective for both volume data compression and visualization [Ballester-Ripoll and Pajarola, 2015; Ballester-Ripoll and Pajarola, 2016; Ballester-Ripoll et al., 2015; Suter et al., 2011; Suter et al., 2013; Suter et al., 2010]. TA methods are essentially a form of higher order singular value decomposition (HOSVD) [de Lathauwer et al., 2000] and like similar methods, such as principal component analysis (PCA), they can perform effective lossy data compression via rank reduction.

Properties of TA methods

The following section (Properties of TA methods) is mostly taken from our publication [Ballester-Ripoll et al., 2018] in which we introduced convolution in the tensor-compressed domain as an effective tool for filtering large volume data[1]*, created in joint work with R. Ballester from the Visualization and MultiMedia Lab. The information in this section is reproduced from the manuscript for completeness, and its attribution is shared by all coauthors.*

TA methods allow for multidimensional data tensors (of dimensionality 3 or higher), such as volume data, to be compactly expressed in terms of multilinear bases and coefficients. By approximating the input data with less complex tensors, TA is becoming a popular framework to cope with the *curse of dimensionality*. TA methods exhibit a number of benefits, such as competitive compression ratios, *rank truncatability* [de Lathauwer et al., 2000], easy progressive reconstruction, etc. Furthermore, this family of techniques has been shown to be similar to or even outperform other frequency domain transform approaches in certain visualization applications (see [Suter et al., 2010; Wu et al., 2008; Wu et al., 2007]).

Two major TA models are relevant in the context of our work: With respect to a volume $\mathcal{A}$, the *canonical* (CP) model describes a rank-R decomposition as a

[1] Please see the article for more details, also on tensor approximation in general.

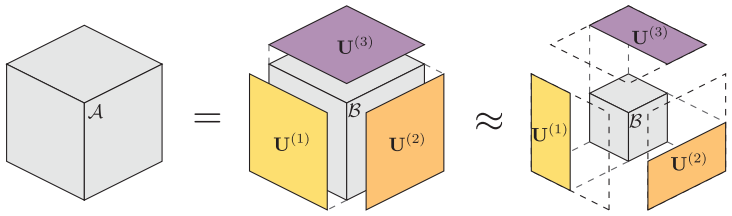
single linear sum over individual vector outer products as

$$\widetilde{\mathcal{A}} = \sum_{r=1}^{R} \lambda_r \cdot \boldsymbol{u}_r^{(1)} \circ \boldsymbol{u}_r^{(2)} \circ \boldsymbol{u}_r^{(3)}. \tag{5.4}$$

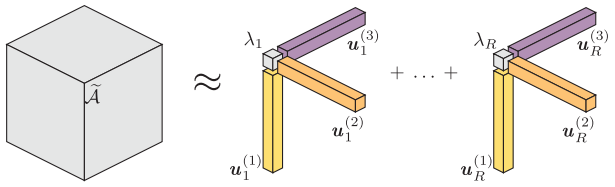
The *Tucker* model [de Lathauwer et al., 2000], on the other hand, written in *tensor-times-matrix notation*, represents a weighted combination over all possible vector outer products as:

$$\begin{aligned} \widetilde{\mathcal{A}} &= \mathcal{B} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}, \\ \widetilde{\mathcal{A}} &= \sum_{r_1} \sum_{r_2} \sum_{r_3} b_{r_1 r_2 r_3} \cdot \boldsymbol{u}_{r_1}^{(1)} \circ \boldsymbol{u}_{r_2}^{(2)} \circ \boldsymbol{u}_{r_3}^{(3)}, \\ \widetilde{a}_{ijk} &= \sum_{r_1} \sum_{r_2} \sum_{r_3} b_{r_1 r_2 r_3} \cdot u_{i\ r_1}^{(1)} \cdot u_{j\ r_2}^{(2)} \cdot u_{k\ r_3}^{(3)}, \end{aligned} \tag{5.5}$$

where $\mathcal{B} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is a *core tensor* of weight coefficients $b_{r_1 r_2 r_3}$, and $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times R_n}$ are the basis *factor matrices* with column vectors $\boldsymbol{u}_{r_n}^{(n)}$ and elements $u_{i\ r_n}^{(n)}$; see also Fig. 5.4.



(a)



(b)

Figure 5.4: *A volume dataset $\mathcal{A}$ can be expressed as its Tucker decomposition (a), which can be approximated via rank reduction. $\mathcal{A}$ can analogously be approximated via a rank-R CP decomposition (b).*

Our work is based on the *Tucker* tensor model which has previously been used for multiscale and multi-resolution volume visualization [Suter et al., 2010; Suter et al., 2011; Suter et al., 2013; Tsai, 2015]. This model has the interesting property that the convolution between two arbitrary tensors can be performed directly on their Tucker factor matrices [Khoromskij and Khoromskaia, 2007; Khoromskij, 2010].

Octree hierarchy

Our system is based on TAMRESH [Suter et al., 2013] and is similarly based on an octree Tucker decomposition of the input volume, which is illustrated in Figure 5.5; in a preprocessing step, the volume is spatially subdivided according to the octree and for each of the resulting nodes, a small core tensor is computed by approximating the corresponding volume data (Figure 5.5(a)). This is performed using the Tucker model (Figure 5.4(a)), yielding a set of factor matrices (Figure 5.5(b)). As we outlined in [Ballester-Ripoll et al., 2018], in this context

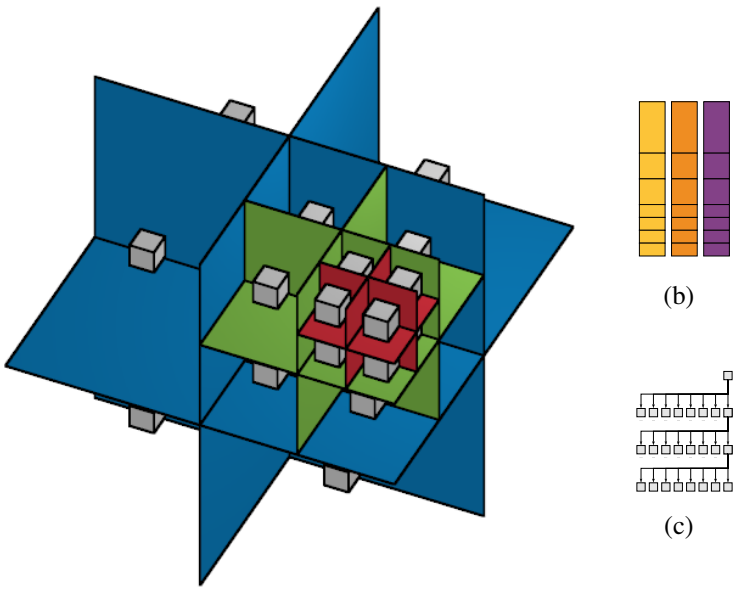


Figure 5.5: *Volume data structures used by our system: octree containing core tensor bricks (a) (the related hierarchy of bricks is illustrated in figure (c)), global set of factor matrices* $\mathbf{U}^{(1\ldots3)}$ *(b).*

the significant difference to the original system is that we consistently use a set of *global* factor matrices, which allows us to apply convolution filters at full spa-

tial resolution. To make effective use of the multi-resolution hierarchy, however, these matrices must be downsampled before rendering, which is a conceptually and computationally simple step (which, however, requires certain preprocessing steps that we further detailed in [Ballester-Ripoll et al., 2018]). It exploits the factor matrices having the property of the rows corresponding to spatial resolution (whereas the columns correspond to the approximated ranks). Downsampling is consequently performed along the columns of the matrices in concert with filtering operations, equalizing the number of rows allocated for each brick and hence reducing spatial resolution. See also Figure 5.10 for an illustration of that process. To avoid artifacts such as discontinuities during filtering and reconstruction, a small number of overlapping values between neighboring bricks is maintained as *border* rows; when a brick has no neighbor, these are empty and filled with zeros (which is also visible in aforementioned figure).

Convolution of Tucker factor matrices

We exploit aforementioned properties of the Tucker model and its associated factor matrices in our system to perform bandwidth-effective convolutions directly in the tensor-compressed domain. As we demonstrated in [Ballester-Ripoll et al., 2018], a volume model can be effectively filtered in the TA-compressed domain by applying a convolution operation to a set of global Tucker factor matrices which correspond to volume bricks at their highest resolution. This is illustrated in Figure 5.6(a): filtering a volume $\mathcal{A}$ with a separable 3-dimensional kernel $\mathcal{G}$ (expressible as outer product $\boldsymbol{g}^{(1)} \circ \boldsymbol{g}^{(2)} \circ \boldsymbol{g}^{(3)}$) is equivalent to convolving the columns of the factor matrices $\mathbf{U}^{(1\ldots3)}$ of the volume's Tucker decomposition with $\boldsymbol{g}^{(1\ldots3)}$. These kernels themselves can also be expressed as a CP decomposition of $\mathcal{G}$ (Figure 5.6(b)).

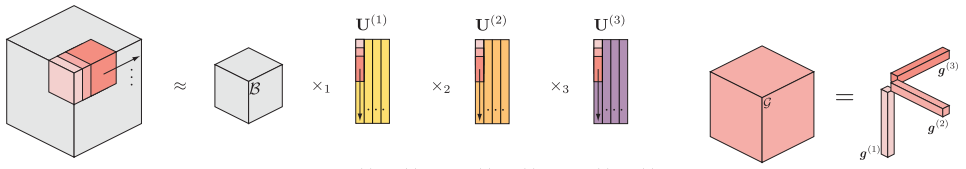


Figure 5.6: *Filtering tensor-approximated volume data: (a) filtering a volume $\mathcal{A}$ with a separable 3-dimensional kernel $\mathcal{G}$ by convolving the factor matrices $\mathbf{U}^{(1\ldots3)}$ of the volume's Tucker decomposition with $\boldsymbol{g}^{(1\ldots3)}$; (b) these can be obtained from $\mathcal{G}$ via CP decomposition.*

This naturally only applies to linear filters and there are several filter operations that cannot be fully expressed in terms of linear convolution. However, these

can often be partially expressed as such, which means that they require an additional pre- or post-processing step. The Sobel operator, e.g., consists of convolution filters for calculating the gradients within a dataset but requires an additional non-linear post-processing step to determine their magnitude.

Comparing filtering approaches

In our system for the brick size B (64 for the used datasets) we typically choose the number of approximated ranks to be $R = \frac{B}{2}$. As the volume has a size of I, it is important to note that $R \ll I$. The global factor matrices only have a size of $3RI$, compared to the total volume size of I^3; filtering and also downsampling these matrices is very inexpensive, both computationally and in terms of memory access: for a separable kernel with size K this amounts to $O(3RIK)$ operations. Apart from this step, tensor-approximated filtering (TAF) further requires a subsequent reconstruction of volume data, however, *at only the resolution that is required* for each potentially visible brick of the volume (see also Figure 5.8). To obtain correct results in the spatial domain, i.e., filtering after decompression before downsampling (FD), on the other hand, requires filtering all I^3 voxels of the volume at full resolution and subsequently downsampling the high-resolution data. This amounts to $O(3I^3K)$ operations for a separable kernel and $O(I^3K^3)$ otherwise.

Since that is prohibitively expensive, another approach is conceivable: filtering after decompression and downsampling (DF). As mentioned previously, this approach has the disadvantage of introducing additional errors. It requires downsampling the used filter kernels, which is often not possible due to the size and nature of the kernel (see also Section 5.2.1). The resulting incorrect filter responses can lead to very different results, depending on the level of detail. Furthermore, as this approach also operates on spatial volume data, it typically requires several stages (1 stage for each of the kernels a convolution filter consists of) and launching these stages creates additional overhead costs (see Section 5.3.1).

The reduction in dimensionality of our TAF method makes filtering large volumes much more scalable: in practice it means accessing and processing a few megabytes of factor matrix data instead of gigabytes of uncompressed or decompressed volume data: E.g., the global factor matrices of our 2048^3 garlic dataset have around 1.5 M entries in total (with 32 bit floating point precision in memory). That is only about 6 MB for the GPU to process, which is a trivial task for modern graphics hardware. Consequently, filtering and downsampling can be performed in typically under a millisecond, even for larger volumes (see Section 5.4). As we showed in [Ballester-Ripoll et al., 2018], directly convolving volume data in the tensor-compressed domain is comparable to convolution of uncompressed data (and subsequently compressing it), in terms of accuracy.

Since filtering is performed in the compressed domain, the filtered volume data still must be reconstructed before it can be rendered. However, as is pointed out in Section 5.4 in more detail: Even with the added cost of reconstruction, our method performs not only more accurate than the DF approach, but is typically faster as well. Considering the requirement of multi-resolution filtering of compressed volume data, we can evaluate the available options as follows from Table 5.1, which highlights the TAF method as a sensible choice for an interactive volume filtering and visualization system. This evaluation reflects both the aforementioned principial properties of each approach, as well as the experimental results described in Section 5.4.

Table 5.1: *Comparison of approaches to interactive volume filtering: after decompression before downsampling (FD), after decompression and downsampling (DF), and tensor-approximated filtering (TAF).*

Approach	**Accurate**	**Fast**
FD	● ● ●	○ ○ ○
DF	○ ○ ○	● ○ ○
TAF	● ● ○	● ● ○

5.3 System Overview

Our system for the interactive filtering and visualization of volumes is based on TAMRESH [Suter et al., 2013]. To cope with large amounts of volume data, the data is tensor-compressed and kept out-of core, i.e, individual volume bricks at different levels of detail are stored in a least-recently-used (LRU) cache in main memory and uploaded and subsequently decompressed in GPU memory, when required for rendering. See Figure 5.7 for an overview of our system.

When the user moves the virtual camera, its view frustum is intersected with the hierarchy of bricks. This hierarchy is then traversed within this intersection until each brick is available in a sufficiently high resolution (Figure 5.8), by projecting the bounding volume of the brick into screen space, or until a predefined budget is reached.

The selection of this potentially visible set happens on the CPU, which uploads any bricks that are currently not available from the LRU cache to the GPU in compressed format. When a brick is not cached, the fetching mechanism is instructed to load it from hard drive. Due to the multi-resolution hierarchy, bricks are always available at a lower resolution for rendering, should loading fail or be delayed. All of the data transfer between hard drive, main memory, and video memory is

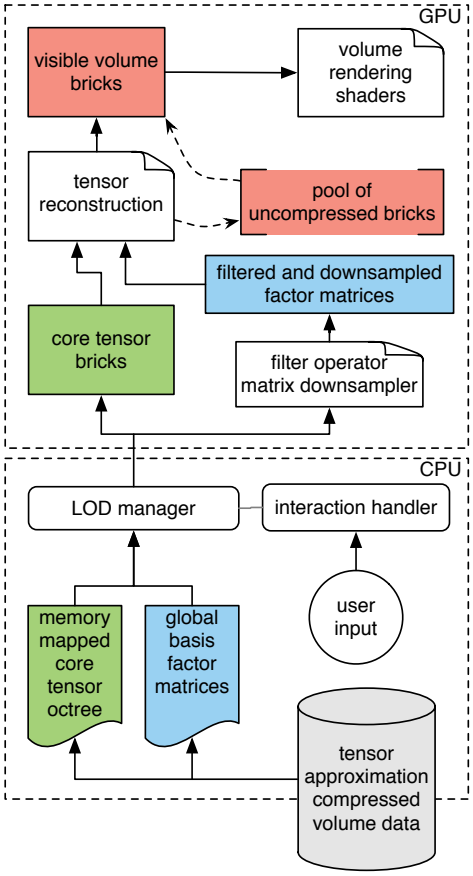


Figure 5.7: *Overview of our volume rendering and filtering system.*

performed asynchronously by two systems, each employing an individual update thread: these are the RAM and GPU Loader, respectively. See Figure 5.9 for an overview of the data flow within those two systems and the Model thread which is ultimately responsible for rendering.

Communication with and between these systems is typically performed asynchronously, via posting commands to a queue that is processed by the respective system's update thread.

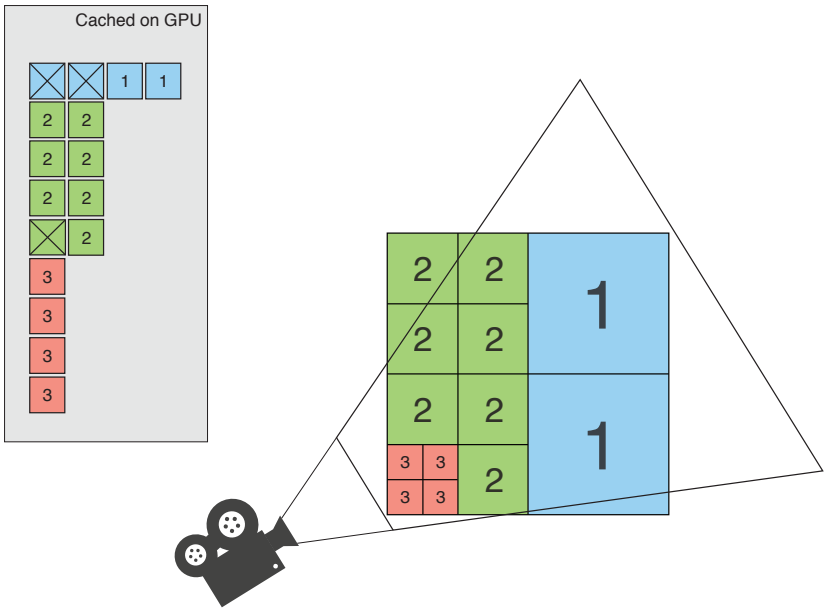


Figure 5.8: *Rendering a multi-resolution hierarchy of volume bricks. A visible set of bricks is determined by the camera's view frustum being intersected with the volume hierarchy on the CPU; bricks within the view frustum are expanded and cached on the GPU for rendering. Higher numbers within the bricks (colored squares) indicate a higher level of detail, crossed-out bricks are available within the cache but currently not visible.*

5.3.1 Filtering Implementation

Filtering is entirely performed on the GPU using *nVidia*'s CUDA platform. A *CUDA kernel* is a program that is executed by sets of threads (*thread blocks*) on the GPU in a data-parallel fashion, once by each of the threads. In the case of the TAF method, this means that two dimensional thread blocks operate on the global factor matrices; in the case of the DF approach, the CUDA kernels are executed by three dimensional thread blocks operating on reconstructed volume data.

For TAF, only a single CUDA kernel is launched and applied to a buffer containing the global factor matrices. At least one CUDA thread is allocated for every matrix column, filtering and downsampling the matrices in an effective data-parallel manner and writing the result to a destination buffer, a *matrix set*. This is illustrated in Figure 5.10; the figure also illustrates how brick sizes within the matrices are affected by downsampling (boundaries between bricks are visible as discontinuities within the matrix data). For the global factor matrices the region allocated for each brick reflects the size of its corresponding octree cell, while the

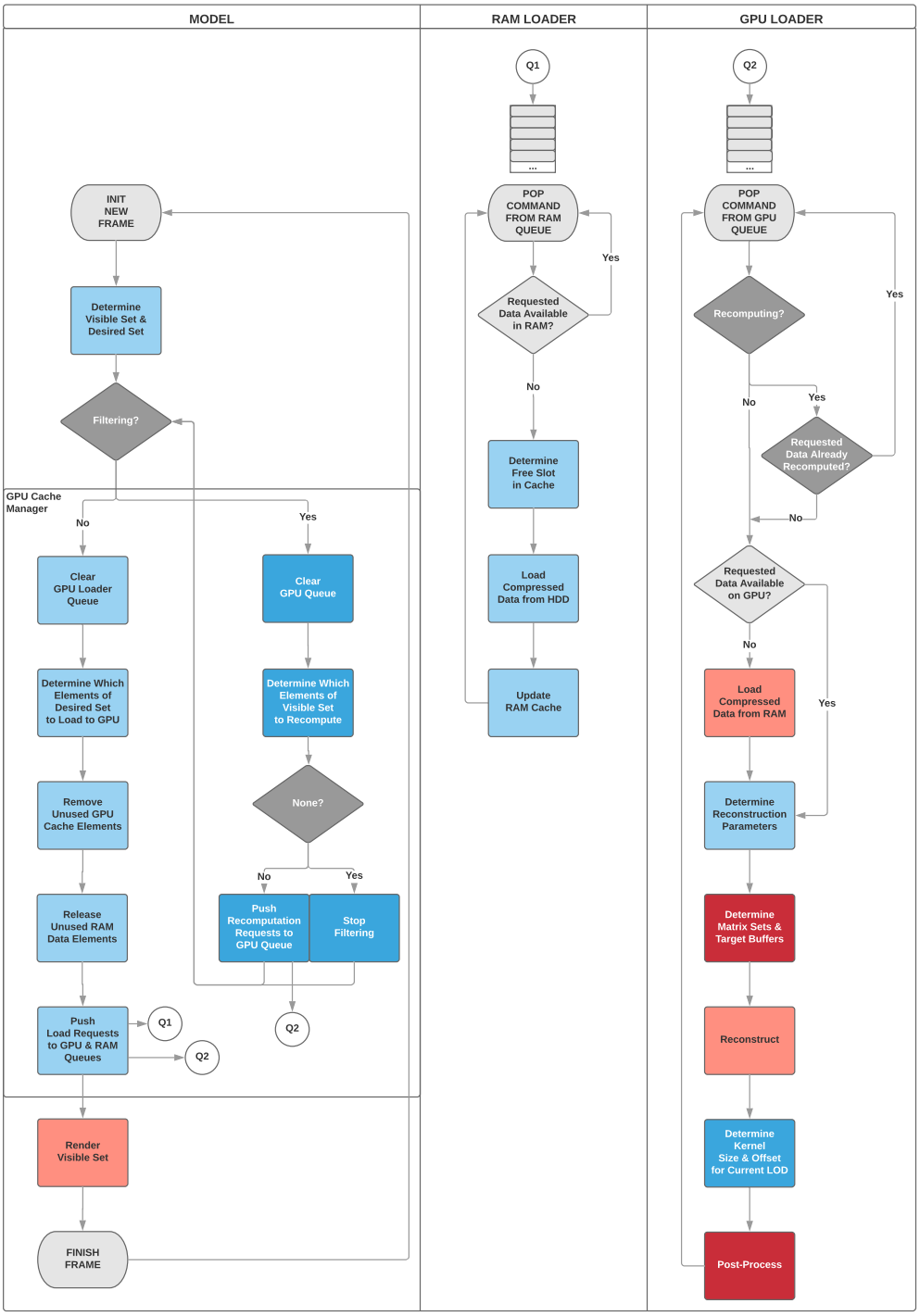


Figure 5.9: *Overview of the data flow in our volume rendering and filtering system (simplified). Processes in blue are performed on the CPU, those in red on the GPU. Additions to the original TAMRESH system are indicated by a darker background and white text.*

downsampled matrix set allocates an equal number of rows for each brick; this relation is also illustrated by Figure 5.5(a).

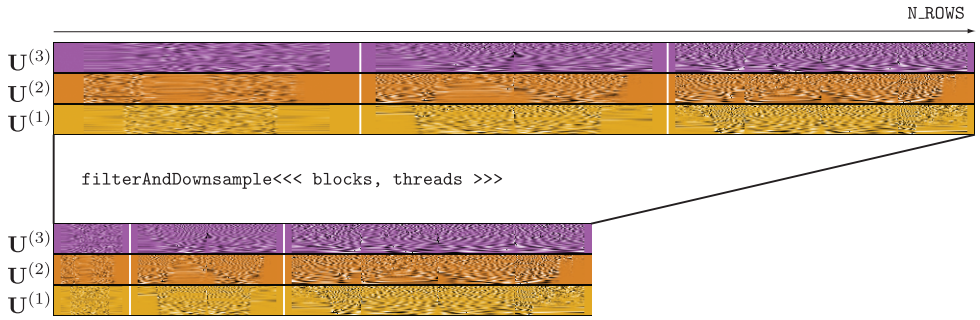


Figure 5.10: *Filtering and downsampling: CUDA kernel applied to the global factor matrices of the Bonsai dataset, high positive values are white, low negative values are black. An identity filter kernel is used, i.e., the matrices are only downsampled. Note that the images have been rotated, i.e., the available* `N_ROWS` *matrix rows of the buffers span from left to right.*

In the case of DF, however, filtering is performed in the post-processing stage on the reconstructed volume data. A filter is then typically applied as a sequence of CUDA kernel launches. In the case of the Sobel operator, e.g., this includes nine CUDA kernel invocations, as illustrated by the following simplified listing:

```
calcFilterXFloat<<< blocks, threads >>>(...kernel1 params...); //∂/∂x A
calcFilterYFloat<<< blocks, threads >>>(...kernel0 params...);
calcFilterZFloat<<< blocks, threads >>>(...kernel0 params...);

calcFilterXFloat<<< blocks, threads >>>(...kernel0 params...);
calcFilterYFloat<<< blocks, threads >>>(...kernel1 params...); //∂/∂y A
calcFilterZFloat<<< blocks, threads >>>(...kernel0 params...);

calcFilterXFloat<<< blocks, threads >>>(...kernel0 params...);
calcFilterYFloat<<< blocks, threads >>>(...kernel0 params...);
calcFilterZFloat<<< blocks, threads >>>(...kernel1 params...); //∂/∂z A.
```

This illustrates how CUDA kernels are launched, by specifying a grid of thread blocks. Three different CUDA parallel programs (one along each axis) are invoked with parameters for two different filter kernels, directly implementing the convolutions listed in Equation 5.3. This assumes that kernel1 calculates partial derivatives via central differences, while kernel0 performs the additional smoothing.

Besides being intrinsically more expensive (as it operates on 3-dimensional, instead of 2-dimensional data), the DF approach also suffers from the overhead of

additional CUDA kernel launches, typically three times as many. This is because TAF can make use of the fact that the three factor matrices are laid out in a single buffer. This buffer can consequently be processed by a single CUDA kernel, hence incurring the overhead of only one CUDA kernel launch. Conversely, it is not practical to process the reconstructed volume data in a similar manner, which would be required to avoid additional launches for DF filtering. Avoiding an additional CUDA kernel launch is also the reason why filtering and downsampling is part of the same operation in our TAF implementation.

Filtering Process

Consistent with [Suter et al., 2013], and as further demonstrated in Section 5.4, our system spends most of the time each frame with rendering, while updating data (including filtering) is a comparably inexpensive task. Since rendering is most expensive during each frame, and data updates are performed asynchronously, we found it necessary to synchronize filtering and rendering tasks, i.e., halt rendering and only continue after filtering has been completed. Otherwise, filtering would be perceived as less responsive: As only a fraction of time each frame is spent on filtering, it would take significantly longer for completion of the filtering operation.

Within our system, filtering of a volume involves the five steps outlined in Figure 5.11. First, rendering is halted. Then, two different transfer functions, one for before and one for after filtering, are selected. This is followed by a filter being selected and subsequently applied. Once the filtering process is finished, rendering is resumed.

Figure 5.11: *Overview of the filtering process.*

Applying a filter, specifically, is a three-stage process that is depicted in Figure 5.12. The figure also illustrates that the filtering subsystem is implemented as a state machine which is controlled via a set of flags and modes. First, the reconstruction mode is set (1), which determines the target buffer and precision for the reconstruction process. Subsequently, post-processing flags are set (2), determining which steps, if any, to perform in the post-processing stage, i.e., after reconstruction has been performed; for the DF approach, most of the work is performed in this stage. For our direct TAF approach, on the other hand, filtering is mostly performed on the full resolution Tucker factor matrices, when the multi-resolution versions of the these matrices are created in step 6.

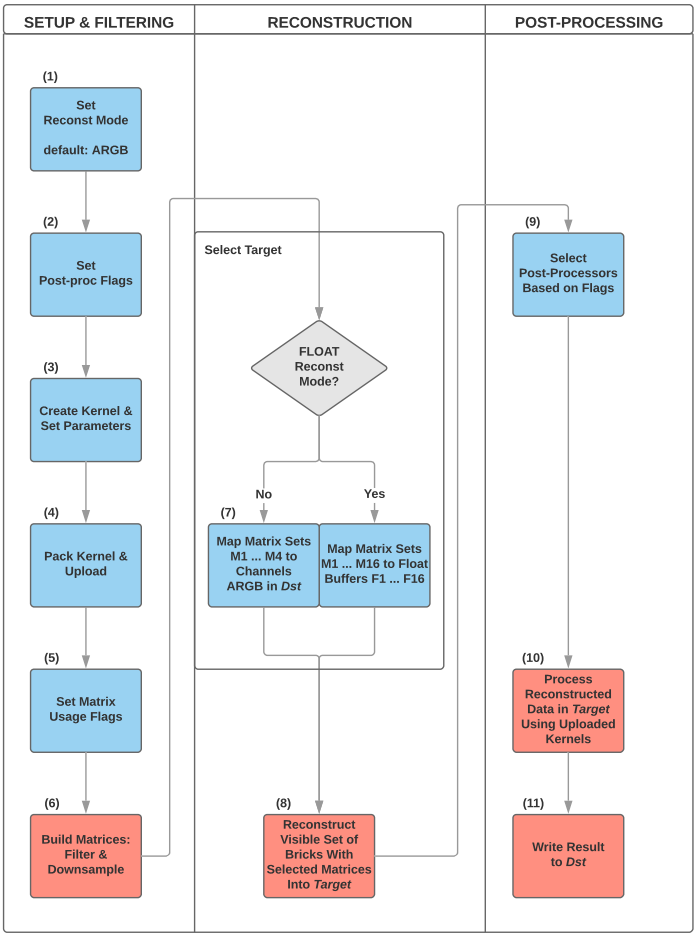


Figure 5.12: *Applying a filter as a three-stage process (simplified). Steps performed by the GPU are colored in blue, steps on the GPU in red.*

After the post-processing flags are set, the required kernels for this filter are created and parameters such as kernel size are set up (3). Subsequently, the kernels are packed into a single buffer and uploaded to the GPU (4). Then, one or more matrix sets are selected. These are each a full set of factor matrices $\mathbf{U}^{(1\ldots3)}$ required to reconstruct all volume bricks at multiple resolutions. Aforementioned selection is performed by setting appropriate matrix usage flags (5), see Algorithm 4 for an example.

The next step is building the selected matrix sets (6). To this end, a set of three kernels $\mathcal{G}_{1\ldots3}$ is mapped to each matrix set $\mathcal{M}$. I.e., each kernel $\mathcal{G}_i$ is mapped to a corresponding factor matrix $\mathbf{U}^{(i)}$, which amounts to using these kernels to filter compressed volume data along its three dimensions. From the set of global factor

Algorithm 4 Implementation of Sobel operator within our system (TAF)

1: SetReconstMode(`RM_FLOAT`)
2: SetPostProcFlags(`PPF_MAGNITUDE_FLOAT`)
3: $\mathcal{H} \leftarrow (1, 2, 1)$
4: $\mathcal{H}' \leftarrow (1, 0, -1)$
5: $\mathcal{S} \leftarrow (\mathcal{H}, \mathcal{H}')$
6: SetKernelSize(3)
7: PackKernels($\mathcal{S}$)
8: UploadKernels()
9: SetMatrixUsageFlags(`MATRIX_SET1 | MATRIX_SET2 | MATRIX_SET3`)
10: BuildMatrices $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

matrices, each selected matrix set is constructed in a single step on the GPU using the specified kernels; if no kernel is specified for a matrix $\mathbf{U}^{(i)}$, an identity kernel is used (see also Figure 5.10).

Next, a reconstruction target is selected, i.e., each matrix set is mapped to a destination for the reconstruction process (7). Depending on the selected *reconstruction mode*, this is either a channel in an RGBA texture (four 8-bit channels) which holds the volume data that is directly used for rendering (which is interpreted as opacity + tint), or one of a set of 16 auxiliary buffers, each with 32-bit floating point precision but only a single channel. These buffers can only hold B^3 values and are suitable for storing intermediate results of multiple stage filters with adequate precision.

Subsequently, the currently visible octree nodes are reconstructed (8) using the earlier created matrix sets, writing into the selected targets and hereby producing TA-filtered volume data that can either be rendered (if reconstructed to the RGBA target) or post-processed, which is typically required for filters that are non-linear but can partially be represented by linear convolution operations.

Based on specified flags, a set of post-processors is selected (9) to perform the final stages of the filter operation. These are small parallel routines (CUDA kernels) executed on the GPU that include operations such as calculating the magnitude of a vector, which is needed for the Sobel operator, or adding the contents of specified buffers and writing the result to a different buffer. DF Filters are mostly implemented in terms of post-processing operations or have a dedicated post-processor (such as the DF Sobel implementation) for reasons of efficiency.

Using aforementioned buffers as source of reconstructed data and for storing intermediate results, and also utilizing the selected kernels that have been uploaded to the GPU and mapped to the active matrix sets, the selected post-processors are executed (10) and finally write the resulting data to the destination buffer for rendering (11).

Aformentioned features of the filtering subsystem can be thought of as practical building blocks that allow the implementation of filters by only specifying the filter kernels themselves and certain flags, including the desired post-processors.

This is best illustrated by the example of the Sobel operator which is implemented as is outlined by Algorithm 4 (TAF) and Algorithm 5 (DF), respectively; the filtering process can be understood by contrasting the two approaches. Most notably, the two implementations only differ in the used post-processing steps and the selection of matrix sets.

Algorithm 5 Implementation of Sobel operator within our system (DF)

1: SetReconstMode(`RM_FLOAT`)
2: SetPostProcFlags(`PPF_DF_SOBEL`)
3: $\mathcal{H} \leftarrow (1, 2, 1)$
4: $\mathcal{H}' \leftarrow (1, 0, -1)$
5: $\mathcal{S} \leftarrow (H, H')$
6: SetKernelSize(3)
7: PackKernels($\mathcal{S}$)
8: UploadKernels()
9: SetMatrixUsageFlags(`MATRIX_SET1`)
10: BuildMatrices()

This is because in the DF case, albeit the filter kernels are set up identically (lines 3-8), filtering is performed entirely in the post-processing step using the provided filter kernels, which is indicated by the flag in line 2. The TAF implementation, conversely, only requests a post-processor that calculates the magnitude of the values that are written to the selected float buffers (line 1) during reconstruction, to perform the final, non-linear step of the Sobel operator. Therefore, as filtering is instead performed on the Tucker factor matrices, matrix usage flags must be set up accordingly (line 9), so that three different matrix sets are built (line 10) and used for reconstruction, one for each partial derivative that the Sobel operator computes. At that step, kernels are mapped to individual matrices, which is illustrated by the matrix argument in line 10: Each row corresponds to one of the three requested matrix sets and each column to one of the factor matrices $\mathbf{U}^{(1\ldots3)}$. The values correspond to a kernel's index within $\mathcal{S}$, i.e., $\mathcal{H}$ is represented by 0, while $\mathcal{H}'$ is represented by 1. The matrix therefore illustrates how $\mathcal{H}$ is used to calculate partial derivatives for each dimension of the dataset,

while also performing a simple smoothing $\mathcal{H}'$. The DF approach, conversely, only requires one (default) matrix set (line 9) and as filtering is performed after reconstruction, it also does not need to map kernels to it; consequently, BuildMatrices is called without any arguments (line 10).

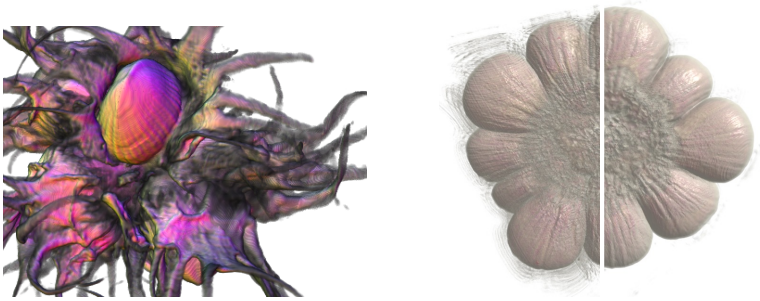
In practice this means, however, that a default kernel will be used for filtering, when matrices are built (see also Figure 5.15); this is currently an identity kernel (of size 1 and value 1). Linear convolution in the tensor-compressed domain is, in fact, computationally so inexpensive (see Section 5.4) that it is a fixed part of our visualization pipeline. Moreover, building matrices, i.e., filtering and downsampling is implemented as a single CUDA program to avoid invocation costs, which makes this approach also more practical.

Multimodal Data

We have further extended our system to be capable of loading, processing and rendering compressed multimodal datasets by supporting several data *channels* for input (loading of bricks) and output (reconstruction and rendering). This can be used to, e.g., assign colors to individual voxels. The internal state machine of our system hereby allows us to flexibly use different available input channels as data sources, and output channels as targets (sinks) for the reconstruction process. This can be used, e.g., for inexpensively pre-calculating a volume's gradients with our TAF method and storing them in the additional channels for further use during rendering (Figure 5.13(a)).

We can also use multimodal datasets for input, which, for example, allows us to load and visualize volume datasets that use color information. In the context of interactive volume filtering, however, this functionality is even more useful when a filter requires non-linearly transformed data as input. The guided filter (Figure 5.13(b)), e.g., has this requirement in our implementation. It is a nonlinear edge-preserving filter [He et al., 2010] that is commonly used for denoising image data, based on a window size w, a damping factor δ, and a guidance image / volume (in our case the input volume itself). Similar to the Sobel operator, we can also partially express this filter in terms of convolutions: as the guided filter requires to calculate the variance of input data, we store an additional squared version of the data in a second input channel and convolve it with a box kernel in the compressed domain, allowing an effective implementation of the guided filter within our system.

Currently, all multimodal data is stored as core tensor bricks with several interleaved channels sharing a single set of Tucker factor matrices as bases. This assumes that all modes of a dataset can be reconstructed with very little error using the same set of factor matrices, an assumption that seems reasonable for various types of data and clearly holds true in the case of input data for the guided fil-



(a) (b)

Figure 5.13: *Example use cases for multimodal data. Multiple target channels holding gradients of the Hazelnut dataset, inexpensively calculated via TAF (a). Guided filter (b) using an additional input channel, applied to Garlic dataset (noisy input data shown on the lefthand side of the figure and the result of filtering on the righthand side), volume filtered with kernel size $K = 17$ and parameters $w = 17$, $\delta = 30$.*

ter. Please see our publication [Ballester-Ripoll et al., 2018] for more information about our implementation of the guided filter, especially regarding its accuracy when compared to naive alternatives.

Kernel Packing

Packing of filter kernels also requires providing them in multiple resolutions: this is needed by the DF approach, with downsampling of a base kernel being a practical solution to achieve this goal. As mentioned in Section 5.2.1, downsampling of filter kernels produces inaccurate results or is often impossible, as in the case of the Sobel operator. A possible alternative could be consistently providing filters in analytical form. However, this would only be feasible for some filter operations and make our system less flexible; it would, e.g., no longer be possible to provide filters as CP decompositions of generic kernels. The kernel packing and downsampling step is therefore a compromise induced by the DF filtering approach, i.e., applying filters in the spatial domain, which is naturally required when comparing our TAF method to alternative methods. It is, however, also a useful tool when filtering cannot be performed entirely in the tensor-compressed domain, e.g., for some stages of the guided filter.

Kernel downsampling is performed during the kernel packing step directly on the CPU, which is very inexpensive since filter kernels are always assumed to be 1-dimensional (leading to only about twice the number of original kernel values to be accessed, when the common approach of averaging with a box filter is used,

i.e., typically well below 100 values in total). This assumption typically holds true for separable filters, whereas non-separable filters can often be decomposed into a set of order 1 tensors that can be used as filter kernels, as explained in Section 5.2.3 (see also [Ballester-Ripoll et al., 2018]).

Unlike in similar multi-resolution data structures, such as MIP maps [Williams, 1983], or the multi-resolution version of the factor matrices used in our system (Figure 5.10), every individual entry in our packed kernels buffer cannot be of size 2^n, but instead requires to be of an uneven size s, specifically

$$s = \begin{cases} 2^n + 1 & \text{for } n \geq 1, \\ 1 & \text{otherwise.} \end{cases} \tag{5.6}$$

This is because a filter kernel's origin, that is used when placing the kernel within a volume, is typically located at the kernel's center. The result is an uneven number of values along each dimension and our system consequently assumes the size of all (1-D) kernels to be uneven. We found it consequently necessary to slightly modify the common approach to building LOD hierarchies via averaging of neighboring samples, as typically employed in MIP maps and similar spatial multi-resolution data structures. Our slightly modified approach is illustrated in Figure 5.14: it differs in the fact that the kernel's central value remains unchanged, while two "1-dimensional image pyramids" are constructed for its neighboring left and right hand side of values, respectively. Although this is naturally suboptimal for preserving the kernel's properties, we chose the described approach due to its simplicity and efficiency.

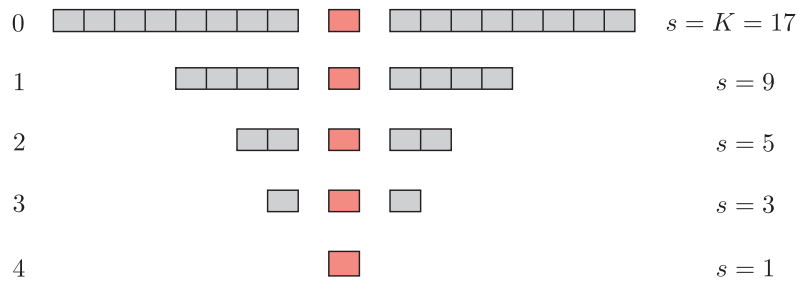


Figure 5.14: *Creating a kernel LOD hierarchy, similar in spirit to a MIP map. The kernel in the example has a base size of $K = 17$ values and is downsampled via averaging neighbors along two pyramids (grey) with a base size of $\lfloor \frac{K}{2} \rfloor$ values each, with a central column, the kernel's origin (red), remaining unchanged. The sizes s of individual kernel LODs are on the right, whereas their indices are listed on the left.*

How these kernel LODs are packed in memory is illustrated in Figure 5.15: levels of detail for two Gaussian kernels, $\mathcal{G}$ and $\mathcal{H}$, as part of a DOG filter are created (a) and packed in a buffer for GPU upload (b); a visual result of our system applying a DOG filter is shown in Figure 5.16.

Each row within the buffer depicted in Figure 5.15(b) corresponds to one individual kernel packed at multiple LOD, in a fashion similar to a 1-dimensional MIP map. As mentioned previously, individual kernel LODs are of uneven size; hence during filtering in the post-processing step (in the spatial domain) they have to be addressed differently. When packed into the buffer, sizes of kernel LODs are defined as

$$s = 2 \left\lfloor \frac{K}{2^{l+1}} \right\rfloor + 1, \tag{5.7}$$

where s denotes the size of a kernel at level l and K denotes its base size.

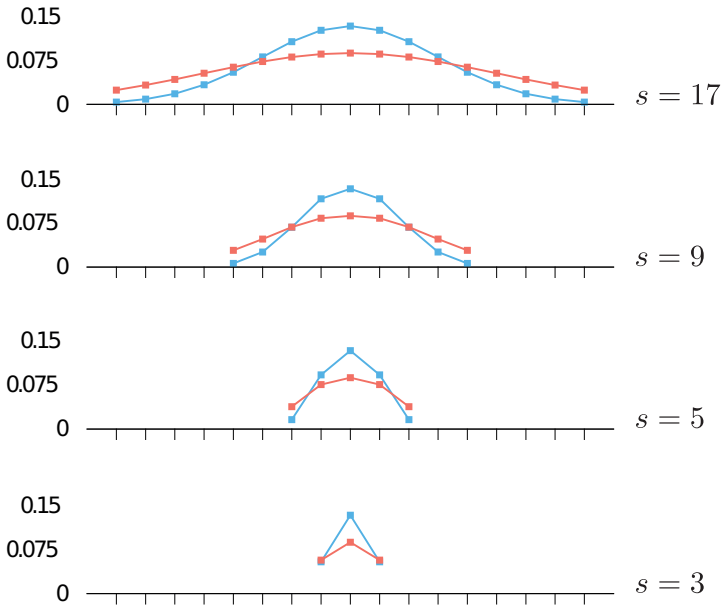
Assuming Equation 5.6 holds true, while addressing a kernel LOD, its offset within the respective row is calculated as

$$o = 2(K-1) \cdot \left(1 - \frac{1}{2^l}\right) + l, \tag{5.8}$$

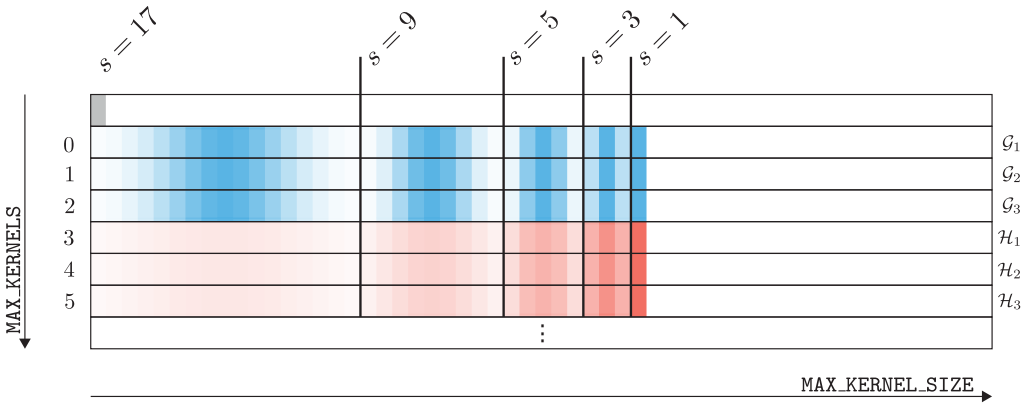
assuming that $\frac{K}{2^l} \geq 1$, i.e., the kernel size at level l is not less than 1.

Kernel downsampling is depicted in Figure 5.15(a) and aforementioned loss of accuracy is illustrated by the fact that the shown curves increasingly less resemble Gaussians, as the kernel size moves from originally $s = K = 17$ values to $s = 3$. These levels of detail are normalized before being applied to volume data and consequently a size $s = 1$ results in an identity kernel. As this would usually (ultimately) functionally break a filter, the resulting level of detail should typically not be used by the application, although it is automatically generated during the packing step, as depicted in Figure 5.15(b).

The DF approach is less accurate in general when compared to TAF (see also [Ballester-Ripoll et al., 2018]); moreover its requirement to use downsampled kernels when processing multi-resolution data introduces additional inaccuracies specifically. However, such an approach still can produce results that are visually very similar to a more correct result obtained via TAF, if the used kernels are rather robust to downsampling, as illustrated by Figure 5.16: It shows a DOG filter applied to the 2048^3 voxels Garlic dataset at 1024^2 pixels screen resolution, hence making use of the downsampled Gaussian kernels in DF mode.



(a) Kernel downsampling



(b) Kernel packing

Figure 5.15: *Downsampling and packing of two Gaussian kernels $\mathcal{G}$ and $\mathcal{H}$ as part of a Difference of Gaussians filter. $\mathcal{G}$ with $\sigma_1 = 3$ and $\mathcal{H}$ with $\sigma_2 = 5$, both are downsampled from a base kernel size of 17 (a) and the resulting multi-resolution kernels are packed in memory (b); lighter colors symbolize lower values. Each row in the buffer represents a kernel, with kernel indices starting at 0. The very first row before index 0 represents the default kernel to be used by the filtering system, in this case an identity kernel consisting of only the value 1 and one level of detail.*

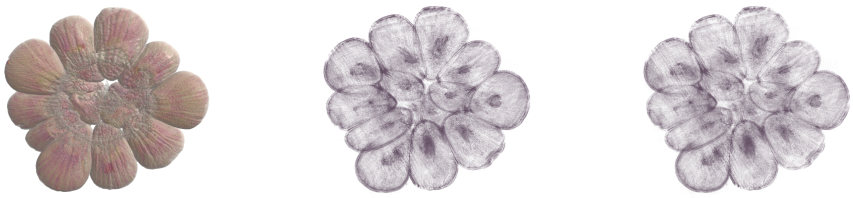


Figure 5.16: *Applying a DOG filter with $\sigma_1 = 1, \sigma_2 = 5, K = 17$ to the Garlic dataset (2048^3) using our system in DF (b) and TAF (c) mode, rendered at 1024^2 pixels resolution. Note that a different transfer function was used after applying the DOG, due to the significant shift in data range that this filter causes.*

5.4 Performance Evaluation

The following section (Performance Evaluation) is mostly taken from our publication [Ballester-Ripoll et al., 2018] in which we introduced convolution in the tensor-compressed domain as an effective tool for filtering large volume data[2], created in joint work with R. Ballester from the Visualization and MultiMedia Lab. The information in this section is reproduced from the manuscript for completeness, and its attribution is shared by all coauthors.

5.4.1 Filtering Performance

To demonstrate the feasibility of our compression domain volume filtering approach, we measured the filtering and reconstruction performance of our proposed TAF compared to spatial domain filtering after downsampling (DF). In our test system, the rendering is temporarily halted whenever filtering is invoked, and resumed again after it is completed and all visible bricks have been reconstructed. This approach displays the full final filtered result as soon as possible, resulting in interactive response times (which depend on the filter complexity). Optionally, the filtering process can also be interleaved with rendering partially filtered and reconstructed results if progressive real-time rendered results are desired.

We rendered 4 datasets at an image resolution of 1024^2 pixels, and applied a *Difference of Gaussians* (DOG) filter to them using the standard deviations of

[2]Please see the article for more details.

$\sigma_1 = 1$ and $\sigma_2 = 5$. The average timings for filtering and reconstruction as performed on the GPU and are reported in Table 5.2, using three different filter sizes of $K = 5, 9, 17$. We used brick border overlaps of size 8. Note that the DOG standard deviations of $\sigma_{1,2}$ do not affect the timings which only depend on the filter size K. We measured and averaged 30 test runs for each configuration, and in order to ensure high-quality rendering for realistic view configurations, the LOD parameter for voxel (brick) selection was chosen such that a voxel is projected onto no more than 2×2 screen pixels.

Table 5.2: *Difference of Gaussians: filtering and reconstruction times (in ms) for all four datasets, comparing post-reconstruction downsampling and filtering (DF) with our compressed domain filtering approach (TAF) for different kernel sizes. The first column indicates the number of processed bricks, each of size $B^3 = 64^3$. Timing values are averaged over 30 test runs using the DOG filter (with $\sigma_1 = 1$ and $\sigma_2 = 5$) of size $K = 5, 9, 17$. No immediate reconstruction is required for the spatial domain filtering DF.*

			Filtering (ms)			
		N. of bricks	**Kernel size**			**Recons-truction (ms)**
			5	9	17	
DF	**Bonsai**	55	114.93	154.56	234.77	–
	Hazelnut	111	224.46	297.59	445.12	–
	Flower	141	257.13	320.38	450.07	–
	Garlic	315	491.19	603.53	775.51	–
TAF	**Bonsai**	55	0.13	0.16	0.16	35.85
	Hazelnut	111	0.18	0.20	0.24	72.33
	Flower	141	0.29	0.32	0.42	91.73
	Garlic	315	0.50	0.61	0.87	204.24

Even though the inaccurate DF approach does not require immediate reconstruction (as spatial filtering is done on the raw bricks), it still requires 3 separable filter passes over the N voxel bricks of size B^3 for each of the S ranks of the filter kernel. In contrast, the TAF approach performs the rank-S approximated DOG filter linearly on the R columns of the factor matrices only, independent of the number N and size B of the voxel bricks. This leads to extremely fast filtering in the compressed domain for virtually any filter size K, and also faster overall final results even after taking reconstruction into account as demonstrated in Table 5.2. We can thus observe that TA reconstruction costs (which dominate the factor matrix filtering by several orders of magnitude) are consistently lower than traditional spatial filtering (DF). In other words, not only does compressed TA-domain filtering offer more accurate results at lower resolution scales, but it is also faster even when accounting for the necessary brick reconstruction time.

For large volumes, accurate spatial filtering of the full resolution volume data followed by downsampling (FD) would be infeasible in real time. First, the dataset is limited in size since its full resolution version must fit into the GPU memory whenever the whole volume is visible. Second, the filtering operation would always entail maximal cost and be especially inefficient for strongly downsampled renderings of zoomed-out views on screen. Having to reconstruct the full resolution in all cases for filtering means that the LOD hierarchy is not exploited at all, which defeats the purpose of using multi-resolution volume rendering in the first place.

To put the performance results in perspective to previous work, in [Treib et al., 2012] the reported average timings for uploading and decompressing a 1024^3-sized scalar field are $1.3s$, without accounting for the subsequent full resolution filtering costs. Our framework, on the other hand, exploits a hierarchical structure to render volume regions at adaptively different resolution levels. The LOD selection coupled with tensor-based compression is able to deliver complete filtered results at times well below half a second. However, performance cannot directly be compared as in [Treib et al., 2012] the volume data has to be fully loaded on the GPU and is only processed after decompression, thus it does not support LOD-based compression domain filtering and rendering.

Tables 5.3 and 5.4 show equivalent experiments for the Sobel operator and guided filter. The former has a fixed kernel size of 3, while for the latter we used again the values 5, 9, and 17. The filtering timings in these cases combine both the convolution in the compressed domain and the necessary post-processing steps thereafter. While these filters are non-linear and more expensive than the DOG counterpart, they are still faster and more accurate than the naive DF.

Fig. 5.17 shows exemplary screenshots of the renderer's state before and after applying the DOG and the guided filter using our method.

Table 5.3: *Sobel operator: filtering and reconstruction times (in ms) for all four datasets*

		N. of bricks	Filtering (ms)	Reconstruction (ms)
DF	**Bonsai**	55	143.53	–
	Hazelnut	111	290.86	–
	Flower	141	369.56	–
	Garlic	315	825.31	–
TAF	**Bonsai**	55	44.84	48.25
	Hazelnut	111	90.44	97.68
	Flower	141	113.97	122.78
	Garlic	315	255.71	272.17

Figure 5.17: *Left column: 4 volumes. Middle column: guided filter result ($w = 5$ and $\delta = 5$ for the Bonsai, $w = 5$ and $\delta = 10$ for the Hazelnut, $w = 9$ and $\delta = 10$ for the Flower, and $w = 17$ and $\delta = 30$ for the Garlic). Right column: Difference of Gaussians ($\sigma_1 = 1, \sigma_2 = 5$; filter kernel size $K = 17$ except for the Bonsai, which used $K = 9$). Note that we updated the transfer function for the DOG, since the filter significantly shifts the data range.*

Table 5.4: *Guided filter: filtering and reconstruction times (in ms) for all four datasets ($\delta = 3000$)*

			Filtering (ms)			
		N. of bricks	Window size			Recons-truction (ms)
			5	9	17	
DF	**Bonsai**	50	240.64	316.76	476.01	–
	Hazelnut	86	401.21	519.61	772.58	–
	Flower	247	1066.04	1320.99	1853.28	–
	Garlic	362	1325.84	1603.21	2020.86	–
TAF	**Bonsai**	50	159.23	196.84	270.87	42.42
	Hazelnut	86	265.49	323.09	441.08	72.87
	Flower	247	728.59	853.54	1105.52	207.72
	Garlic	362	955.71	1087.20	1290.67	306.55

5.4.2 Rendering Performance

On the Vranx graphics workstation (Section 2.4.1), utilizing one GPU, we evaluated the interactive performance of our system with the three largest volumes, including the 8 GB Garlic (see Figs. 5.18 to 5.20).

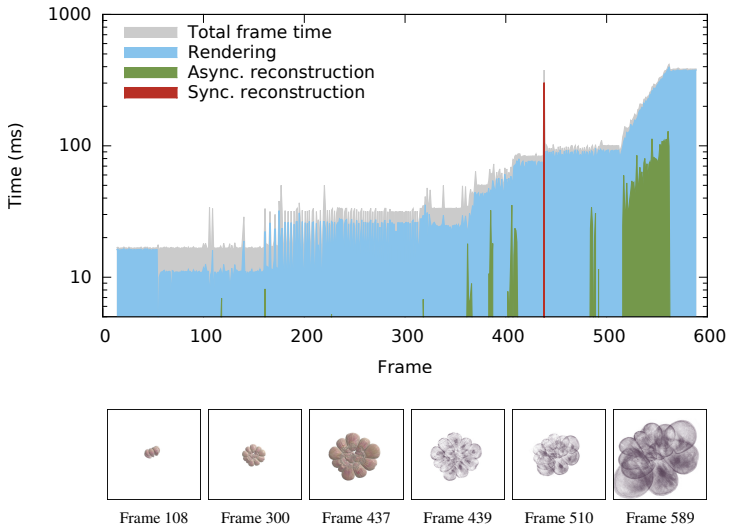


Figure 5.18: *Timings (log-scale) for the interactive visualization of the Garlic with a DOG filter operation in-between. We show the time required per frame for rendering (blue),* asynchronous *reconstruction for LOD updates (green) and* synchronized *reconstruction due to filtering (red), as well as the total time required to produce each frame (gray). We used kernel size $K = 17$, and $\sigma_1 = 1, \sigma_2 = 5$.*

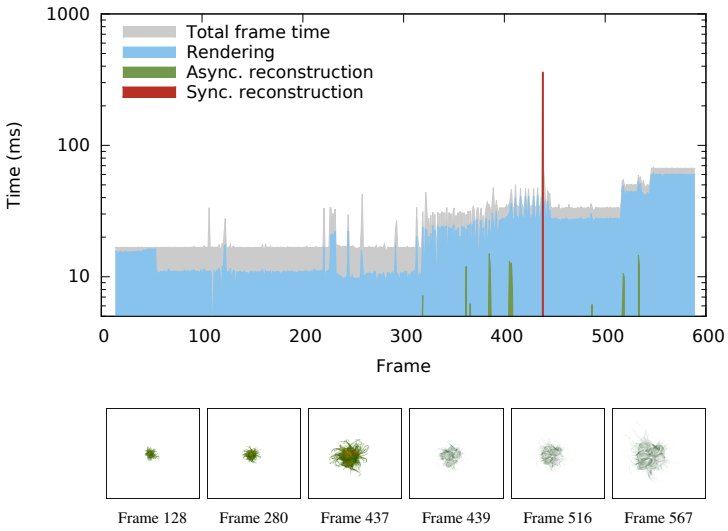


Figure 5.19: *Timings (log-scale) for the interactive visualization of the Hazelnut with a Sobel filtering in-between.*

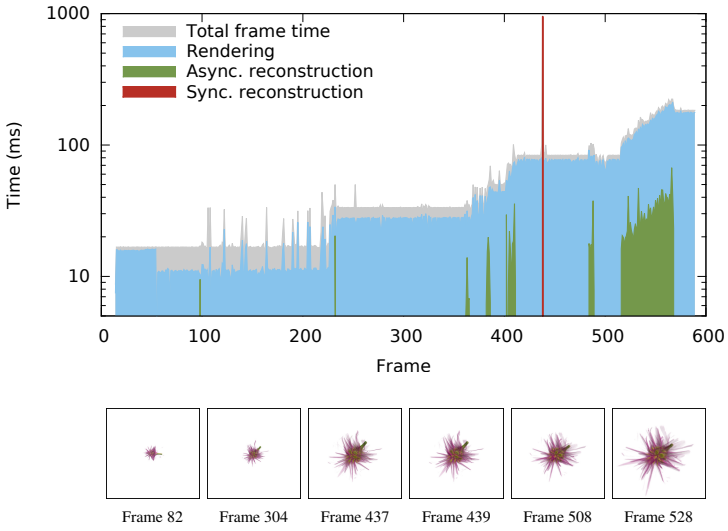


Figure 5.20: *Timings (log-scale) for the interactive visualization of the Flower with a guided filtering in-between ($w = 5, \delta = 1000$).*

Using the Tin utility (see Section 2.3.2), we recorded and used a camera path starting from a zoomed-out view and progressing towards a more detailed close-up, with a filtering step applied at frame 437. The timings in Fig. 5.18 to Fig. 5.20

reveal that for interactive visual exploration the vast majority of time per frame is spent on rendering (volume ray casting), while the time required for incremental reconstruction of updated LOD volume bricks is much less significant. This behavior is consistent with the related TAMRESH system [Suter et al., 2013]. Note that the total frame time is less than the sum of rendering and reconstruction, since we exploit *asynchronous* incremental LOD volume brick updates and reconstruction as in [Suter et al., 2013].

A *synchronized* reconstruction of all visible volume bricks is needed after a filter operation has been invoked, as opposed to normal asynchronous LOD updates and rendering during viewpoint changes. Nevertheless, such reconstruction delays the next displayed frame only by typically a fraction of a second. In Figs. 5.18 to 5.20 this is indicated by the reconstruction cost peaks in red. Thus, after the user initiates a filter operation, the system can still react and update the rendered image of the complete filtered result at an interactive rate. In the given example a frame update time of about one third of a second is the only delay a user experiences for applying a (complex) filter operation, before interactive rendering of the now filtered volume resumes. Note that all overhead costs are reflected by the total frame time (gray), including initialization and finishing of a frame, CPU and GPU data management and transfer; and in the case of filtering, setting up and uploading filter kernels, etc.

5.5 Conclusion

We presented a system capable of applying convolutions to multi-resolution volume data directly in the tensor-compressed domain. The compact Tucker tensor representation we use allows us to accurately filter the data at full spatial resolution before it is reconstructed and rendered. Such convolutions of compressed data are performed on the GPU very effectively and typically require less than a millisecond, even for large volumes.

While convolution using this representation is computationally very inexpensive, this is not the case for reconstruction and rendering. However, data is only reconstructed and rendered at the resolution determined by the target device. Moreover, reconstruction costs must be put into perspective: Rendering costs are much more significant than the costs for reconstruction, which is often necessary in any case, as modern volume visualization systems typically rely on compressed representations of data.

Our approach is typically faster and produces more accurate results than the other viable option for interactive visualization of multi-resolution volume data, which is filtering after decompression and downsampling (DF), that requires downsampling also of filter kernels (which is often not possible without compromising

the essential properties of the kernels). No viable alternative, although more accurate, is filtering after decompression before downsampling (FD), as it defies the purpose of using a multi-resolution hierarchy for volume rendering and would be prohibitively expensive. Conversely, for our method, the entire process of filtering, reconstruction, and rendering, can be accurately performed at interactive frame rates and on volumes that are too large to fit in memory, i.e., out of core.

Nonlinear filters are only supported by our system insofar they can be partially represented via convolutions. Our results demonstrate that this can still be both viable and effective, as working in the tensor-compressed domain, i.e., with a set of compact Tucker factor matrices, typically requires significantly less operations than directly processing (potentially large) subvolumes. We showed that such a "hybrid" approach, filtering data both in the spatial and the tensor-compressed domain in multiple stages, can still be faster and more accurate than approaches that directly filter multi-resolution volume data in the spatial domain. We demonstrated this for the guided filter and the Sobel operator, which both can be partially expressed via convolution operations.

CHAPTER 6

CONCLUSIONS

6.1 Summary

The scalable visualization of large datasets is a challenge with multiple dimensions, several of which were addressed by this thesis. Appropriate visualization systems need to be both data and performance scalable to accommodate increasingly large datasets, and be capable of supporting higher throughputs and lower latencies.

With this end in view, such a system can be scaled to accommodate more workload. Among the less repeatable strategies to accomplish this goal are optimizing I/O in general and making use of compact data representations at multiple levels of resolution to optimize the use of memory and bandwidth, which have been addressed by this thesis. A specific example is performing filter operations in the tensor-compressed domain, which is very efficient, both in terms of bandwidth usage and computationally. As demonstrated by our interactive volume filtering system, such operations can typically be performed in less than a millisecond, also for large (≈ 8.6 billion voxels) volumes. Even when considering the additional cost of reconstructing the compressed data, our approach is superior to alternatives, both in terms of performance and accuracy.

More significantly, a visualization system can be scaled vertically by increasing the capacity of a single node, and even more horizontally by utilizing additional nodes. Related strategies tend to allow more incremental and repeatable scaling but the distributed nature of the resulting parallel rendering systems cre-

ates further challenges, such as additional communication and synchronization overheads as well as load imbalances. To build scalable visualization systems, overcoming load imbalances is critical, especially in the context of interactive scenarios that demand low latency but can exhibit unpredictable load patterns. A similar issue is that the amount of available resources might fluctuate, if a machine is not dedicated to only a single task, which can be the case in the context of virtualization.

We therefore developed a scalable and flexible rendering task partitioning approach and an associated node affinity model based on data locality which allow fine-grained implicit dynamic load balancing. Unlike explicit methods, our load balancer based on dynamic work packages does not need to rely on frame-to-frame coherence (and on statistics about previous frames) when a new frame is being rendered and tasks are assigned to render nodes. Instead, nodes themselves pull work packages from a server-based queue, while the frame is being rendered, making the method more flexible and adaptive. We analyzed our method in terms of performance and scalability and demonstrated that it often performs better than traditional load balancing methods. This is especially the case when facing load that varies strongly between frames; we could also demonstrate better data scalability for work packages under such conditions. Moreover, our method tends to exhibit better data scaling behavior when using a higher number of packages, although this incurs an initial overhead.

We implemented our load balancing method based on work packages as a compound for the Equalizer parallel rendering platform and consequently evaluated performance and data scalability of all Equalizer compounds most relevant to interactive visualization. Tree and Package equalizer compounds usually exhibited the best performance in these experiments, in which we also demonstrated that Tree equalizer copes well with heterogeneous render nodes when it can a-priori bias its load balancing.

Finally, we also extended the set of tools available to develop and profile parallel rendering applications and to systematically and automatically evaluate the performance of scalable visualization systems: to this end we introduced additional utility classes and functions into the Equalizer parallel rendering platform and its underlying libraries.

6.2 Future Work

Although this thesis has addressed several of the aspects related to the vast topic of scalable visualization, there naturally still remain many issues that should be investigated in future. These include the following:

- *Automatic selection of work package count*

  Instead of requiring the user to manually specify this setting in the Equalizer configuration file, the system should be able to automatically deduce a suitable number of work packages to be created for each frame. This could be based on the number of available render nodes and the size of the dataset to be visualized, which might be difficult to implement in a meaningful way without further increasing the coupling between the Equalizer platform and the application using it. Another possibility could be for the system to very slowly optimize the number of work packages until a suitable value is found. The difficulty here would lie in avoiding to re-introduce issues such as oscillations in the task partitioning process, that the work packages method avoids by not relying on frame-to-frame coherence.

- *A DMA-based system*

  The Equalizer platform is based on Collage, which is a networking library for building flexible distributed systems. Consequently, communication, including data transmission, is mostly based on the technique of message passing. However, a modern interconnect, such as Infiniband, provides remote direct memory access (RDMA), which is currently not used directly but could reduce communication overhead. Moreover, many other modern system components allow direct memory access in some form, most importantly for scalable visualization: graphic cards and hard drives. This is often accomplished via highly optimized asynchronous API calls. Allowing these possibilities to play a central part in a parallel rendering system will likely become increasingly important and could help to further increase a system's throughput. This might be accomplished by introducing an additional abstraction layer for handling direct memory accesses, partially replacing and complementing Collage.

- *Experiments on larger scale*

  While this thesis was focused on parallel rendering systems of relatively small scale (up to 10 nodes), our experiments regarding data scalability of Equalizer compounds seem to suggest that the simple FCFS Package equalizer might have the potential for driving much more massive visualization setups, presumably due to its low overhead. Experiments on a much larger scale are required to see how this compound and the Equalizer system actually behave under such conditions and whether underlying algorithms can be further optimized for massive scale.

- *Better downsampling*

  We explained that we use a box filter as lowpass before downsampling factor matrices, as it is common practice and computationally very efficient. However, there is no necessity to do so. The GPU-based filtering and downsampling process that we described is computationally so inexpensive that it could also be implemented using a much better filter (e.g., Lanczos) for that purpose, without significantly decreasing our method's performance.

- *Horizontal scaling of TAF system*

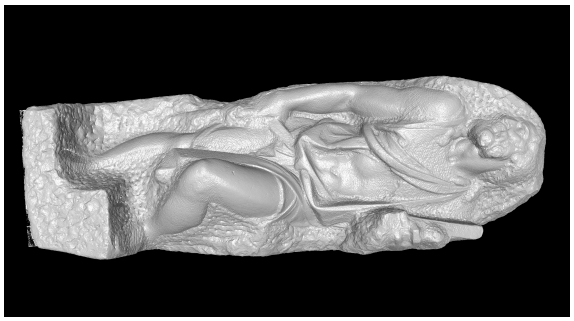
  Our system for interactively filtering large volume data using tensor approximation is also based on the Equalizer platform which facilitates scaling the system, e.g., on a visualization cluster. However, additional experiments are required to thoroughly investigate especially the data scalability of the system in such a context.

APPENDIX A

DESCRIPTION OF DATASETS

A.0.1 Polygonal Models

StMatthew



Laser scan of Michelangelo's St. Matthew.

Size	≈ 372.8 M triangles
Origin	Digital Michelangelo Project, Stanford University, `https://graphics.stanford.edu`

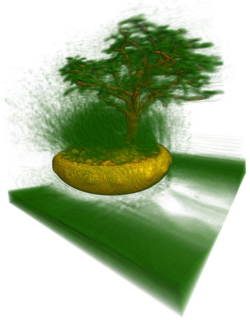
David

Laser scan of Michelangelo's David (1mm).

Size	≈ 56.2 M triangles
Origin	Digital Michelangelo Project, Stanford University, `https://graphics.stanford.edu`

A.0.2 Volumes

Bonsai



CT scan of a bonsai tree.

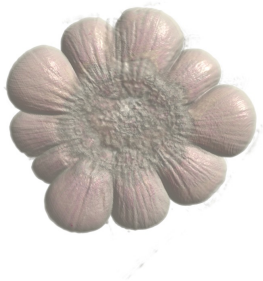
Size	256^3 voxels
Precision	unsigned, 8 bit
Origin	S. Roettger, University of Stuttgart, `http://www.volvis.org`

Flower

μCT scan of a dried flower (leucadendron rubrum).

Size	1024^3 voxels
Precision	unsigned, 8 bit
Resolution	60 μm per voxel
Origin	S. Suter, University of Zurich, `http://www.ifi.uzh.ch/en/vmml`

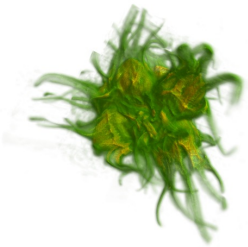
Garlic



μCT scan of garlic bulb.

Size	2048^3 voxels
Precision	unsigned, 8 bit
Resolution	35 μm per voxel
Origin	J.L. Alatorre, R. Ballester-Ripoll, D. Steiner, University of Zurich, `http://www.ifi.uzh.ch/en/vmml`

Hazelnut



μCT scan of dried hazelnuts.

Size	512^3 voxels
Precision	unsigned, 8 bit
Resolution	148 μm per voxel
Origin	S. Suter, University of Zurich, `http://www.ifi.uzh.ch/en/vmml`

BIBLIOGRAPHY

[Abraham et al., 2004] Abraham, F., Celes, W., Cerqueira, R., and Campos, J. L. (2004). A load-balancing strategy for sort-first distributed rendering. In *Proceedings SIBGRAPI*, pages 292–299.

[Allard et al., 2002] Allard, J., Gouranton, V., Lecointre, L., Melin, E., and Raffin, B. (2002). NetJuggler: Running VR Juggler with multiple displays on a commodity component cluster. In *Proceeding IEEE Virtual Reality*, pages 275–276.

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings American Federation of Information Processing Societies Joint Computer Conference*, volume 30, pages 483–485.

[Ballester-Ripoll and Pajarola, 2015] Ballester-Ripoll, R. and Pajarola, R. (2015). Lossy volume compression using Tucker truncation and thresholding. *The Visual Computer*, pages 1–14.

[Ballester-Ripoll and Pajarola, 2016] Ballester-Ripoll, R. and Pajarola, R. (2016). Tensor decomposition methods in visual computing. In *IEEE Visualization Tutorials*.

[Ballester-Ripoll et al., 2018] Ballester-Ripoll, R., Steiner, D., and Pajarola, R. (2018). Multiresolution volume filtering in the tensor compressed domain.

IEEE Transactions on Visualization and Computer Graphics, 24(10):2714–2727.

[Ballester-Ripoll et al., 2015] Ballester-Ripoll, R., Suter, S. K., and Pajarola, R. (2015). Analysis of tensor approximation for compression-domain volume visualization. *Computers & Graphics*, 47:34–47.

[Balsa Rodríguez et al., 2014] Balsa Rodríguez, M., Gobbetti, E., Iglesias Guitián, J. A., Makhinya, M., Marton, F., Pajarola, R., and Suter, S. K. (2014). State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum*, 33(6):77–100.

[Bethel et al., 2013] Bethel, E. W., Prabhat, P., Byna, S., Rübel, O., Wu, K. J., and Wehner, M. (2013). Why high performance visual data analytics is both relevant and difficult. *Visualization and Data Analysis 2013*, doi:10.1117/12.2010980.

[Bhaniramka et al., 2005] Bhaniramka, P., Robert, P. C. D., and Eilemann, S. (2005). OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126.

[Bierbaum and Cruz-Neira, 2003] Bierbaum, A. and Cruz-Neira, C. (2003). ClusterJuggler: A modular architecture for immersive clustering. In *Proceedings Workshop on Commodity Clusters for Virtual Reality, IEEE Virtual Reality Conference*.

[Bierbaum et al., 2001] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., and Cruz-Neira, C. (2001). VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality*, pages 89–96.

[Cederman and Tsigas, 2008] Cederman, D. and Tsigas, P. (2008). On dynamic load balancing on graphics processors. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 57–64.

[Cho et al., 2012] Cho, Y., Kim, M., and Park, K. S. (2012). LOTUS: Composing a multi-user interactive tiled display virtual environment. *The Visual Computer*, 28(1):99–109.

[Correa et al., 2002] Correa, W. T., Klosowski, J. T., and Silva, C. T. (2002). Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96.

[Crockett, 1995] Crockett, T. W. (1995). Parallel rendering. NASA CR-195080, Langley Research Center. See also ICASE Report 95-31.

[Cruz-Neira et al., 1992] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. (1992). The cave: Audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72.

[de Lathauwer et al., 2000] de Lathauwer, L., de Moor, B., and Vandewalle, J. (2000). On the best rank-1 and rank-($R_1, R_2, ..., R_N$) approximation of higher-order tensors. *SIAM Journal of Matrix Analysis and Applications*, 21(4):1324–1342.

[DeFanti et al., 2011] DeFanti, T. A., Acevedo, D., Ainsworth, R. A., Brown, M. D., Cutchin, S., Dawe, G., Doerr, K.-U., Johnson, A., Knox, C., Kooima, R., Kuester, F., Leigh, J., Long, L., Otto, P., Petrovic, V., Ponto, K., Prudhomme, A., Rao, R., Renambot, L., Sandin, D. J., Schulze, J. P., Smarr, L., Srinivasan, M., Weber, P., and Wickham, G. (2011). the future of the cave. *Central European Journal of Engineering*, 1(1):16–37.

[DeFanti et al., 1989] DeFanti, T. A., Brown, M. D., and McCormick, B. H. (1989). Visualization: Expanding scientific and engineering research opportunities. *Computer*, 22(8):12–25.

[Doerr and Kuester, 2011] Doerr, K.-U. and Kuester, F. (2011). CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332.

[Dudak et al., 2016] Dudak, J., Zemlicka, J., Karch, J., Patzelt, M., Mrzilkova, J., Zach, P., Hermanova, Z., Kvacek, J., and Krejci, F. (2016). High-contrast x-ray micro-radiography and micro-ct of ex-vivo soft tissue murine organs utilizing ethanol fixation and large area photon-counting detector. *Scientific Reports*, 6(1), doi:10.1038/srep30385.

[Eckhardt et al., 2014] Eckhardt, D., Haardtd, M., Jackson, I., Banks, G., and Kerrisk, M. (2014). Open(2). `http://man7.org/linux/man-pages/man2/open.2.html`.

[Eilemann et al., 2009] Eilemann, S., Makhinya, M., and Pajarola, R. (2009). Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452.

[Eilemann and Pajarola, 2007] Eilemann, S. and Pajarola, R. (2007). Direct send compositing for parallel sort-last rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*.

[Eilemann et al., 2018] Eilemann, S., Steiner, D., and Pajarola, R. (2018). Equalizer 2.0 - convergence of a parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, doi:10.1109/TVCG.2018.2870822.

[Engel et al., 2006] Engel, K., Hadwiger, M., Kniss, J. M., Rezk-Salama, C., and Weiskopf, D. (2006). *Real-Time Volume Graphics*. AK Peters.

[Erol et al., 2011] Erol, F., Eilemann, S., and Pajarola, R. (2011). Cross-segment load balancing in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50.

[Eyescale Software GmbH, 2008] Eyescale Software GmbH (2008). Load Equalizer. `http://www.equalizergraphics.com/scalability/loadEqualizer.html`.

[Febretti et al., 2014] Febretti, A., Nishimoto, A., Mateevitsi, V., Renambot, L., Johnson, A., and Leigh, J. (2014). omegalib: A multi-view application framework for hybrid reality display environments. In *Proceedings IEEE Virtual Reality*, pages 9–14.

[Febretti et al., 2013] Febretti, A., Nishimoto, A., Thigpen, T., Talandis, J., Long, L., Pirtle, J., Peterka, T., Verlo, A., Brown, M., Plepys, D., Sandin, D., Renambot, L., Johnson, A., and Leigh, J. (2013). Cave2: A hybrid reality environment for immersive simulation and information analysis. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 8649.

[Fogal et al., 2010] Fogal, T., Childs, H., Shankar, S., Krüger, J., Bergeron, R. D., and Hatcher, P. (2010). Large data visualization on distributed memory multi-GPU clusters. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on High-Performance Graphics*, pages 57–66.

[Friendly, 2006] Friendly, M. (2006). A brief history of data visualization. In Chen, C., Härdle, W., and Unwin, A., editors, *Handbook of Computational Statistics: Data Visualization*, volume III. Springer-Verlag, Heidelberg.

[Garcia and Shen, 2002] Garcia, A. and Shen, H.-W. (2002). An interleaved parallel volume renderer with PC-clusters. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization*, pages 51–60.

[Gebara et al., 2015] Gebara, F. H., Hofstee, H. P., and Nowka, K. J. (2015). second-generation big data systems. *Computer*, 48(1):36–41.

[Grosso et al., 1996] Grosso, R., Ertl, T., and Aschoff, J. (1996). Efficient data structures for volume rendering of wavelet-compressed data. In *Proceedings Winter School of Computer Graphics*. Computer Society Press.

[Gustafson, 1988] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

[Haber and McNabb, 1990] Haber, R. B. and McNabb, D. A. (1990). visualization idioms: A conceptual model for scientific visualization systems. *Visualization in scientific computing*, 74:93.

[He et al., 2010] He, K., Sun, J., and Tang, X. (2010). Guided image filtering. In *European Conference on Computer Vision: Part I*, pages 1–14.

[Heirich and Arvo, 1998] Heirich, A. and Arvo, J. (1998). A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1-2):57–68.

[Hui et al., 2009] Hui, C., Xiaoyong, L., and Shuling, D. (2009). A dynamic load balancing algorithm for sort-first rendering clusters. In *Proceedings IEEE International Conference on Computer Science and Information Technology*, pages 515–519.

[Humphreys et al., 2000] Humphreys, G., Buck, I., Eldridge, M., and Hanrahan, P. (2000). Distributed rendering for scalable displays. In *Proceedings ACM/IEEE Conference on Supercomputing*.

[Humphreys et al., 2001] Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., and Hanrahan, P. (2001). WireGL: A scalable graphics system for clusters. In *Proceedings Annual Conference on Computer Graphics and Interactive Techniques*, pages 129–140.

[Johnson et al., 2006] Johnson, A., Leigh, J., Morin, P., and Van Keken, P. (2006). GeoWall: Stereoscopic visualization for geoscience research and education. *IEEE Computer Graphics and Applications*, 26(6):10–14.

[Jones et al., 2004] Jones, K., Danzer, C., Byrnes, J., Jacobson, K., Bouchaud, P., Courvoisier, D., Eilemann, S., and Robert, P. (2004). SGI® OpenGL Multipipe™ SDK User's Guide. Technical Report 007-4239-004, Silicon Graphics.

[Khoromskij, 2010] Khoromskij, B. N. (2010). Fast and accurate tensor approximation of a multivariate convolution with linear scaling in dimension. *Journal of Computational and Applied Mathematics*, 234(11):3122–3139.

[Khoromskij and Khoromskaia, 2007] Khoromskij, B. N. and Khoromskaia, V. (2007). Low rank Tucker-type tensor approximation to classical potentials. *Central European Journal of Mathematics*, 5(3):523–550.

[Korch and Rauber, 2004] Korch, M. and Rauber, T. (2004). A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47.

[Levoy et al., 2000] Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. (2000). The digital Michelangelo project: 3D scanning of large statues. In *Proceedings ACM SIGGRAPH*, pages 131–144. ACM SIGGRAPH.

[Li et al., 1996] Li, P. P., Duquette, W. H., and Curkendall, D. W. (1996). RIVA: A versatile parallel rendering system for interactive scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):186–201.

[Lippert et al., 1997] Lippert, L., Gross, M., and Kurmann, C. (1997). Compression domain volume rendering for distributed environments. In *Proceedings EUROGRAPHICS*, pages 95–108. also in Computer Graphics Forum 16(3).

[Molnar et al., 1994] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. (1994). A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32.

[Moloney et al., 2007] Moloney, B., Weiskopf, D., Möller, T., and Strengert, M. (2007). Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 45–52.

[Moreland, 2013] Moreland, K. (2013). a survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378.

[Moreland et al., 2001] Moreland, K., Wylie, B. N., and Pavlakos, C. J. (2001). Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 85–92. IEEE.

[Müller et al., 2006] Müller, C., Strengert, M., and Ertl, T. (2006). Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 59–66.

[Neal et al., 2011] Neal, B., Hunkin, P., and McGregor, A. (2011). Distributed OpenGL rendering in network bandwidth constrained environments. In Kuhlen, T., Pajarola, R., and Zhou, K., editors, *Proceedings Eurographics Conference on Parallel Graphics and Visualization*, pages 21–29. Eurographics Association.

[Nie et al., 2005] Nie, W., Sun, J., Jin, J., Li, X., Yang, J., and Zhang, J. (2005). A dynamic parallel volume rendering computation mode based on cluster. In *Proceedings Computational Science and its Applications*, volume 3482 of *Lecture Notes in Computer Science*, pages 416–425.

[NVIDIA Corporation, 2017] NVIDIA Corporation (2017). CUDA runtime API. `https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`.

[Reda et al., 2013] Reda, K., Febretti, A., Knoll, A., Aurisano, J., Leigh, J., Johnson, A., Papka, M., and Hereld, M. (2013). visualizing large, heterogeneous data in hybrid-reality environments. *IEEE Computer Graphics and Applications*, 33(4):38–48.

[Samanta et al., 2000] Samanta, R., Funkhouser, T., Li, K., and Singh, J. P. (2000). Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 97–108.

[Samanta et al., 1999] Samanta, R., Zheng, J., Funkhouser, T., Li, K., and Singh, J. P. (1999). Load balancing for multi-projector rendering systems. In *Proceedings Eurographics Workshop on Graphics Hardware*, pages 107–116.

[Seo et al., 2012] Seo, D., Tomizato, F., Toda, H., Uesugi, K., Takeuchi, A., Suzuki, Y., and Kobayashi, M. (2012). Spatial resolution of synchrotron x-ray microtomography in high energy range: Effect of x-ray energy and sample-to-detector distance. *Applied Physics Letters*, 101(26):261901.

[Staadt et al., 2003] Staadt, O. G., Walker, J., Nuber, C., and Hamann, B. (2003). A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *Proceedings Eurographics Workshop on Virtual Environments*, pages 261–270.

[Steiner et al., 2016] Steiner, D., Paredes, E. G., Eilemann, S., and Pajarola, R. (2016). Dynamic work packages in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 89–98.

[Sterling et al., 1995] Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., and Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press.

[Suter et al., 2013] Suter, S., Makhynia, M., and Pajarola, R. (2013). TAMRESH: Tensor approximation multiresolution hierarchy for interactive volume visualization. *Computer Graphics Forum*, 32(3pt2):151–160.

[Suter et al., 2011] Suter, S. K., Iglesias Guitián, J. A., Marton, F., Agus, M., Elsener, A., Zollikofer, C. P., Gopi, M., Gobbetti, E., and Pajarola, R. (2011). Interactive multiscale tensor reconstruction for multiresolution volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2135–2143.

[Suter et al., 2010] Suter, S. K., Zollikofer, C. P., and Pajarola, R. (2010). Application of tensor approximation to multiscale volume feature representations. In *Proceedings Vision, Modeling and Visualization*, pages 203–210.

[Sutter, 2005] Sutter, H. (2005). The free lunch is over: a fundamental turn toward concurrency in software. *Dr Dobb's Journal*, 30(3).

[Szalay and Gray, 2006] Szalay, A. and Gray, J. (2006). science in an exponential world. *Nature*, 440(7083):413–414.

[Treib et al., 2012] Treib, M., Bürger, K., Reichl, F., Meneveau, C., Szalay, A., and Westermann, R. (2012). Turbulence visualization at the terascale on desktop PCs. *IEEE Transactions on Visualization and Computer Graphics (Proc. Scientific Visualization 2012)*, 18(12):2169–2177.

[Tsai, 2015] Tsai, Y.-T. (2015). Multiway K-clustered tensor approximation: Toward high-performance photorealistic data-driven rendering. *ACM Transactions on Graphics*, 34(5):157:1–15.

[Valdetaro et al., 2014] Valdetaro, A., Raposo, A., and Elias, P. (2014). modular distributed visualization and collaboration for a real-time 3d visualizer. In *Proceedings International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 32–39.

[Weinstock and Goodenough, 2006] Weinstock, C. and Goodenough, J. (2006). on system scalability. Technical Report CMU/SEI-2006-TN-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[Williams, 1983] Williams, L. (1983). Pyramidal parametrics. In *Proceedings ACM SIGGRAPH*, pages 1–11. ACM.

[Wu et al., 2008] Wu, Q., Xia, T., Chen, C., Lin, H.-Y. S., Wang, H., and Yu, Y. (2008). Hierarchical tensor approximation of multidimensional visual data. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):186–199.

[Wu et al., 2007] Wu, Q., Xia, T., and Yu, Y. (2007). Hierarchical tensor approximation of multidimensional images. In *Proceedings IEEE International Conference in Image Processing*, volume 4, pages IV–49–IV–52.

[Wulf and McKee, 1994] Wulf, W. and McKee, S. A. (1994). Hitting the memory wall: Implications of the obvious. Technical report, University of Virginia, Charlottesville, VA, USA.

[Yeo and Liu, 1995] Yeo, B.-L. and Liu, B. (1995). Volume rendering of DCT-based compressed 3D scalar data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43.

CURRICULUM VITAE

Personal Information

Name	David Steiner
Date of birth	April 08, 1986
Place of birth	Austria

Education

2015	M.Sc. in Informatics Multimodal and Cognitive Systems	University of Zurich
2010	M.Sc. in Engineering Digital Media	University of Applied Sciences Upper Austria
2008	B.Sc. in Engineering Media Technology and Design	University of Applied Sciences Upper Austria

Publications

Conference Publications

D. Steiner, E. G. Paredes, S. Eilemann, R. Pajarola, *Dynamic Work Packages in Parallel Rendering*, Eurographics Symposium on Parallel Graphics and Visualization, 2016.

B. Bürgisser, D. Steiner, R. Pajarola, *bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum*, Eurographics Education Papers, 2017.

Journal Articles

R. Ballester-Ripoll, D. Steiner, R. Pajarola, *Multiresolution Volume Filtering in the Tensor Compressed Domain*, IEEE Transactions on Visualization and Computer Graphics, 24(10), 2018.

S. Eilemann, D. Steiner, R. Pajarola, *Equalizer 2.0 - convergence of a parallel rendering framework*, IEEE Transactions on Visualization and Computer Graphics, to appear.

Misc

D. Steiner, E. G. Paredes, S. Eilemann, F. Erol, and R. Pajarola. *Dynamic Work Packages in Parallel Rendering*, Technical Report IFI-2015.04, University of Zurich, 2015.